Objective-C

The Ultimate Guide

SUFYAN BIN UZAYR

CRC Press
Taylor & Francis Group

Objective-C

Objective-C is a general-purpose, object-oriented programming language that extends the C programming language with Smalltalk-style messaging. While simultaneously developing for OS X and iOS, Objective-C's capabilities have been bolstered by the inclusion of a dynamic runtime and assistance for object-oriented programming.

Objective-C: The Ultimate Guide walks developers and coders through a straightforward and practical method of learning the Objective-C programming language. This book discusses the basics in brief, and then moves on to more advanced and detailed exercises to help readers quickly gain the required knowledge. The focus in this book remains on writing optimized and well-structured code in Objective-C.

Key Features:

- Follows a hands-on approach and offers practical lessons and tutorials related to Objective-C
- Discusses Objective-C using real world industry concepts
- Includes at-length discussion of Objective-C concepts to help build robust knowledge

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Objective-C
The Ultimate Guide

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

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

ISBN: 9781032413211 (hbk)
ISBN: 9781032413198 (pbk)
ISBN: 9781003357506 (ebk)

DOI: 10.1201/9781003357506

Typeset in Minion Pro
by KnowledgeWorks Global Ltd.

For Dad

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Contents

About the Author

Sufyan bin Uzayr is a writer, coder, and entrepreneur with 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. He 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 numerous 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

CHAPTER 1

Crash Course in Objective-C

IN THIS CHAPTER

- What is Objective-C
- Major Concepts
- Advantages and Disadvantages
- Syntax and Code Basics
- Additional Info

Objective-C language is a general-purpose, object-oriented programming language that extends the C programming language with Smalltalk-style messaging. Apple's primary programming language for the OS X and iOS operating systems and their associated APIs, Cocoa, and Cocoa Touch. This book will walk you through a straightforward and practical method of learning the Objective-C programming language.

Programming in Objective-C is a general-purpose programming language. Although it is not unique to any platform or system, use it to construct a wide range of frameworks. Programming in Objective-C adds communications features to the language C.

Objective-C is one of the primary programming languages used by Apple for the iOS platform and is used to develop mobile apps for this platform. Being the superset of the C programming language, it allows

DOI: 10.1201/9781003357506-1

developers to be more detail-oriented and accommodating of objects and other programming languages.

Numerous programming languages exist. Three programming languages are now in high demand: Swift, Objective-C , and C++. Let's examine the fundamental distinctions between Objective-C and the other two programming languages:

OBJECTIVE-C SPECIFICS

Objective-C is excellent for memory management; there are available compilers that can turn Objective-C code into static code analysis, which the language then uses to distinguish important information from "trash."

The most crucial aspect of Objective-C to understand is that it is very object-oriented. Using this additional language, you can move graphs and modify files, but it is crucial to understand its limits to comprehend its benefits.

Objective-C, a programming language created in the 1980s, retains many features used in iOS-specific mobile app development. While there has been no breakthrough that enables Objective-C to be utilized on all platforms, Objective-C is compatible with C and other languages for iOS apps.

Objective-C competes with Swift, a more recent iOS programming language. Several discussions over whether programming language provides superior iOS mobile application development outcomes.

WHY OBJECTIVE-C?

For several reasons, the Objective-C programming language is selected. It's an object-oriented language, first and foremost. Object-oriented approaches are required to give the type of capability seen in the Cocoa frameworks. Second, since Objective-C is an extension of ANSI C, existing C applications are converted to utilize the software frameworks without sacrificing any of the efforts that went into their creation. Because Objective-C includes C, you receive all of the advantages of C while dealing with it.

You may select whether to use object-oriented programming methods (for example, to create a new class) and when to use procedural programming techniques (define a structure and some functions instead of a class).

Furthermore, Objective-C is a very basic programming language. It has a simple, clear syntax that is simple to pick up.

Object-oriented programming poses a high learning curve to new recruits with its self-conscious vocabulary and focuses on abstract design. A well-structured language, such as Objective-C, may make becoming an expert object-oriented programmer considerably easier.

Objective-C is exceptionally dynamic when compared to other C-based object-oriented languages. For usage at runtime, the compiler saves a lot of information about the objects themselves. Decisions taken at compile time are deferred until after the program has been executed. Objective-C programs have a lot of flexibility and power because of their dynamism. It provides two major advantages that are difficult to get with other ostensibly object-oriented languages.

Objective-C has an open, dynamic binding approach that can allow a basic interactive user interface design. Messages are not always bound by the receiver's class or method name. Therefore a software framework may enable users to make decisions at runtime and provide developers a creative latitude in their design. (Terms like dynamic binding, message, class, and receiver will be defined later in this text.)

Dynamism allows for the creation of advanced development tools. It's feasible to construct tools that monitor, intervene, and disclose the underlying structure and activity of Objective-C programs using an interface to the runtime system, which offers access to information about running apps.

DIFFERENTIATING OBJECTIVE-C VERSUS SWIFT

Swift is the programming language that Apple introduced in June 2014. Objective-C has all the flaws one would anticipate from a language derived from C. To differentiate keywords and types from C types, Objective-C prefixes new keywords with @. Swift is not based on C. Therefore, it may combine all Objective-C types and object-related keywords and extract their many @ symbols.

In addition to other modern programming languages, Swift code nearly resembles natural English. This readability makes it easier for existing programmers from JavaScript, Java, Python, C#, and C++ to adopt Swift as part of their toolchain, in contrast to the painful experience that Objective-C programmers had.

DIFFERENTIATING OBJECTIVE-C FROM C++

Wilkerson states that C++ was introduced in 1979 to combine objects and instance methods with the original C programming language. Objective-C language was founded on the belief that object-oriented programming

would be more productive and successful for big software projects; several senior experts see this as the cause for C++'s widespread acceptance in the years that followed. According to programming community experts, C++ is the language used to create most current desktop applications. As a result of its popularity, many frameworks and libraries have been developed to extend C++ for functions such as high-performance graphics, audio digital signal processing, and user interface design.

Both languages are derived from C, but they are two entirely distinct languages. Objective-C relies heavily on its runtime library to handle inheritance and polymorphism, while C relies heavily on compile-time choices for transmission. Nevertheless, in C++, the focus is often on compile-time conclusions. C++ is a middle-level programming language that runs on several cross-platform operating systems including Windows, UNIX, Mac OS, etc. Objective-C, in contrast, is a general-purpose, object-oriented programming language that Apple uses in its operating systems and Cocoa APIs, etc.

UPSIDES AND DOWNSIDES OF OBJECTIVE-C

Upsides

- Compatibility with both C++ and Objective-C++
- Effective attributes such as method swizzling
- More dependable assistance in coding Binary Frameworks.

Downsides

- Since Objective-C is built upon C, namespacing is required. All classes inside an Objective-C program must be globally unique. Therefore, there is a practice of prefixing class names to avoid conflict. This is why 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 lack of strict type make it harder to identify and resolve issues.
- The syntax of the language is tedious and complex.

FACTORS TO CONSIDER

Objective-C application development may be expedited and is an excellent approach to adding object-based functionality to an application. There are many important considerations about this superset language:

Maintenance is essential; this pertains to upgrading Objective-C developed applications. Although the language is dated, it is not outdated. It merely takes minimal maintenance to remain current.

Less adaption is necessary as many APIs still have a lot to catch up for Swift-developed applications. This suggests that Objective-C may be simpler to deploy to iOS mobile applications.

Object handling made simpler Apple is all about having an object network. Using Objective-C, these objects are readily movable.

Objective-C may simplify iOS mobile applications in all Apple iOS devices, including smartphones and tablets.

OBJECTIVE-C OVERVIEW

Objective-C supports object-oriented programming, along with the four object-oriented development pillars.

- Encapsulation
- Data hiding
- Inheritance
- Polymorphism

Example:

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

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

   NSLog (@"hello everyone");
   [pool drain];
   return 0;
}
```

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

SETUP OF THE OBJECTIVE-C ENVIRONMENT

LOCAL ENVIRONMENT CONFIGURATION

If we want to create our own environment for the Objective-C programming language, we must install Text Editor and The GCC Compiler on our computer.

EDITOR OF TEXT

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

On various operating systems, the name and version of the text editor may differ. Notepad, for example, will be used on Windows, and use vim or vi on both Windows and Linux or UNIX.

Source files are files that we produce with our editor that contain program source code. Objective-C source files are commonly named with the extension ".m."

Before starting our programming, make sure we have one text editor in place, and we have enough knowledge to develop a computer program, store it in a file, compile it, and eventually run it.

GCC COMPILER

Our program's source code is written in a source file and is human readable. It must "compiled" into machine code before our CPU can run the program as instructed.

Use this GCC compiler to transform our source code into an executable application. We're assuming we're familiar with programming language compilers.

The GCC compiler is available for free on various systems, and the technique for installing it on those platforms is detailed 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
```

The following command is used to configure package dependencies:

```
$ 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 unlock all of Objective-functionality. The package is obtained by visiting http://wwwmain.gnustep.org/resources/downloads.php

Now we must navigate to the downloaded location and unpack the file using:

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

Now we must navigate to the folder GNUStep-startup, which is created by using:

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

The construction procedure must then be configured.

```
$. /configure
```

Then we may construct by

```
$ make
```

We must eventually create the environment by

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

We have a helloeveryone.m Objective-C file that looks like this:

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

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

   NSLog (@"hello everyone");
   [pool drain];
   return 0;
}
```

Now we can build and execute an Objective-C file, say helloeveryone.m, by using cd to get to the file's location and then doing the following steps:

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

MAC OS INSTALLATION

If we're running Mac OS X, the simplest method to get GCC is to download Apple's Xcode development environment and follow the straightforward installation instructions. Once we've installed Xcode, we'll be able to utilize the GNU C/C++ compiler.

Xcode is presently being downloaded at https://developer.apple.com/xcode/.

WINDOWS INSTALLATION

To run an Objective-C program on Windows, we must first install MinGW and GNUStep Core. Download both at: https://www.gnu.org/software/gnustep/windows/installer.html.

We must first install the MSYS/MinGW System package. The GNUstep Core package then is installed and both of which have a self-explanatory Windows setup.

Then, go to Start -> All Programs -> GNUstep -> Shell to utilize Objective-C and GNUstep.

Navigate to the helloeveryone.m folder.

We may use to compile the code.

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

STRUCTURE OF THE OBJECTIVE-C PROGRAM

Before we examine the fundamental building blocks of the Objective-C programming language, let's look at a bare minimum Objective-C program structure that we can use as a reference in the later chapters.

OBJECTIVE-C EXAMPLE OF HELLO EVERYONE

A simple Objective-C program consists of the following components:

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

Consider a simple code that prints the words "Hello Everyone."

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

@interface SampleClass:NSObject
```

```
- (void)sampleMethod;
@end

@implementation SampleClass

- (void)sampleMethod {
   NSLog(@"Hello, Everyone \n");
}

@end

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

Let's examine several components of the code above:

- The first line of the code, #import <Foundation/Foundation.h>, is a preprocessor instruction that instructs an Objective-C compiler to import the Foundation.h file before real compilation.
- The following line @interface SampleClass:NSObject demonstrates interface creation. It inherits the foundation class for all objects, NSObject.
- (void)sampleMethod; demonstrates how to define a method.
- The following line, @end, signifies the end of an interface.
- The following line demonstrates how to implement the interface SampleClass.
- The sampleMethod is implemented in the following line, (void) sampleMethod.
- The next line @end signifies the conclusion of an implementation.
- The following line, int main(), is the program's main function, where execution starts.

- The next line /*…*/ will be disregarded by the compiler and has been included to provide additional program comments. Therefore, these lines are known as comments in the program.
- The following line NSLog(…) is an Objective-C function that causes the message "Hello, Everyone" to appear on the screen.
- The next line return 0; concludes the main() function and returns the number 0.

BASIC SYNTAX IN OBJECTIVE-C

OBJECTIVE-C TOKENS

A token is a keyword, an identifier, a constant, a string literal, or a symbol in an Objective-C program. For instance, the following Objective-C statement is made up of six tokens:

```
NSLog(@"Hello, Everyone \n");
```

Individual tokens are referred to as:

```
NSLog
@
  (
    "Hello, Everyone \n"
)
;
```

SEMICOLONS;

A semicolon is a statement terminator in an Objective-C program. That is, a semicolon must follow each sentence. It denotes the end of a single logical entity.

For example, consider the following two statements:

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

COMMENTS

Comments are similar to help text in our Objective-C application and are disregarded by the compiler. As illustrated below, they begin with/* and end with the characters */.

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

There can be no comments within comments, and they cannot be within a string or character literals.

IDENTIFIERS

An Objective-C identifier recognizes a variable, function, or other user-defined items. An identifier starts with a letter A to Z, a to z, or an underscore and ends with zero or more letters, underscores, or numbers (0 to 9).

Punctuation characters such as @, $, and per cent are not permitted within identifiers in Objective-C. Objective-C is a computer language that is case sensitive. Thus, in Objective-C, Manpower and manpower are two different IDs. The following are some instances of appropriate identifiers:

```
Kohd       lara    bac   move_name  b_213
name50   _temp   j     b23b9      retVal
```

KEYWORDS

The following is a list of reserved terms in Objective-C. These reserved terms are not permitted to be used as constant, variable, or other identifier names.

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

WHITESPACE IN OBJECTIVE-C

A blank line has simply whitespace, potentially with a remark, and is completely ignored by an Objective-C compiler.

In Objective-C, whitespace refers to blanks, tabs, newline characters, and comments. Whitespace divides one section of a statement from

another and allows the compiler to determine where one element, such as int, ends, and the next element starts in a statement. As a result, in the following sentence

```
int ages;
```

For the compiler to distinguish between int and age, there must be at least one whitespace character (typically a space). In contrast, the following statement,

```
fruit = grapes + apples; // get total fruit
```

There are no whitespace characters required between fruit and =, or between = and grapes. However, we are allowed to include any for readability purposes.

DATA TYPES IN OBJECTIVE-C

Data types are a comprehensive framework used in the Objective-C programming language for declaring variables or functions of various sorts. A variable's type dictates how much storage space it takes up and how the bit pattern recorded is interpreted.

Objective-C types can be classed as follows:

Sr. No.	Types and Description
1	Basic types They are arithmetic types divided into two categories: (a) integer types and (b) floating-point types.
2	Enumerated types They are arithmetic types once more, and they are used to construct variables that can only be allocated discrete integer values throughout the program.
3	The type void The type specifier void signifies that there is no accessible value.
4	Derived types Among them are pointer types, array types, structure types, union types, and function types.

The aggregate types are the array and structure types combined. A function's type indicates the type of the function's return value.

TYPES OF INTEGERS

The table below contains information on standard integer types, including 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

The sizeof operator can determine the precise size of a type or variable on a specific platform. The phrase sizeof(type) returns the object or type's storage size in bytes. The following is an example of determining the size of an int type on any computer.

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

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

TYPES OF FLOATING-POINT

The table below contains information on typical float-point types, including storage sizes, 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 includes macros that let us use these values and other information about the binary representation of real numbers in our applications. The following example will output the storage space occupied by a float type and its range values.

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

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

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).

VARIABLES IN OBJECTIVE-C

A variable is just the 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 the underscore character. Start it 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). It is a type 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 sorts of variables, which we will discuss in later chapters, such as Enumeration, Pointer, Array, Structure, Union, etc. Let us look at fundamental variable types in this section.

OBJECTIVE-C VARIABLE DEFINITION

In Objective-C, a variable definition tells the compiler where and how much storage to generate for the variable. A variable description describes a data type and includes a list of one or more variables of that type, as seen below.

```
type variablelist;
```

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    c, d, e;
char   k, kh;
float  g, salary;
double q;
```

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

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 variablename = value;
```

Example:

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

For variables declared without an initializer, variables with static storage duration are automatically initialized with NULL (all bytes have the value 0). In contrast, the initial value of all other variables is indeterminate.

OBJECTIVE-C VARIABLE DECLARATION

A variable declaration assures the compiler that there is at least one variable with the specified type and name, allowing the compiler to continue with further compilation without requiring more information. A variable declaration only has significance at the time of compilation; the compiler requires the actual variable declaration when linking the program.

A variable declaration is essential when you are utilizing many files. You specify your variable in one of the files accessible at the time of program linking. To define a variable at any location, you will utilize the extern keyword. Variables are declared several times in Objective-C programs, but they can only be defined once per file, function, or code block.

Example: Try out the following example, in which variables are declared at the top but defined and initialized inside the main function.

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

// Variable-declaration:
extern int x, y;
extern int z;
extern float f;

int main () {
  /* variable-definition: */
  int x, y;
  int z;
  float f;

  /* actual-initialization */
  x = 10;
  y = 20;

  z = x + y;
  NSLog(@"The value of z : %d \n", z);

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

  return 0;
}
```

The same idea applies to function declarations, where we offer a function name at the time of declaration, and its actual definition is provided afterward. The following example uses a C function, and as we may know, Objective-C also supports C style functions.

```
// function-declaration
int func();

int main() {
   // function-call
   int k = func();
}

// function-definition
int func() {
   return 0;
}
```

OBJECTIVE-C lvalues AND rvalues

In Objective-C, there are two types of expressions.

- lvalue: Expressions that refer to a memory location are called "lvalue" expressions. An lvalue can be on either the left or right side of an assignment.
- rvalue: The word rvalue refers to a data value stored in memory at some address. An rvalue is an expression that cannot be assigned a value; therefore, it can occur on the right but not the left side of an assignment.

Variables are lvalues and can thus appear on the left side of an assignment. Because numerical literals are rvalues, they cannot be allocated and cannot appear on the left-hand side. The following is a correct statement:

```
int k = 20;
```

However, the following is not a legitimate statement and would result in a compile-time error.

CONSTANTS IN OBJECTIVE-C

The constants denote values that the program cannot modify during its execution. These values are also known as literals.

Any fundamental data type may include integer constants, floating constants, character constants, and string literals. Additionally, there are enumeration constants.

The constants are processed identically to regular variables, except their values are not altered once they are defined.

INTEGER LITERALS

An integer literal may be a decimal, octal, or hexadecimal constant. The base or radix is specified by a prefix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

A literal integer may also include a suffix consisting of U and L, which stand for unsigned and long, respectively. Write the suffix with uppercase or lowercase letters and in any sequence.

The following are instances of integer literals:

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

An integer part, a decimal point, a fractional part, and an exponent part comprise a floating-point literal. Floating-point literals can be represented in either decimal or exponential form.

When representing in decimal form, including the decimal point, exponent, or both; when representing in exponential form, include the integer portion, fractional part, or both. e or E introduces the signed exponent.

The following are some instances of floating-point literal:

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

CHARACTER CONSTANTS

Character literals, such as 'x,' are contained in single quotes and may be kept in a simple char variable.

A character literal in Objective-C can be a simple character (for example, 'x'), an escape sequence (for example, '\t'), or a universal character (for example, 'u02C0').

Certain letters in C have unique significance when followed by a backslash and signify things like newline (\n) or tab (\t). A list of some of these escape sequence codes is as follows.

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

The following is an example of a few escape sequence characters.

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

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

STRING LITERALS

Double quotes “” are used to surround string literals or constants. Characters in a string are comparable to character literals in that they are plain characters, escape sequences, and universal characters. We can divide an extensive line into numerous lines using string literals and whitespaces.

Here are some string literal instances. The strings in all three variants are identical.

```
"hello, sweetie"

"hello, \

sweetie"

"hello, " "s" "sweetie"
```

CONSTANT DEFINITION

In C, there are two straightforward ways to define constants.

- Using #define preprocessor
- Using const keyword

#define Preprocessor

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

```
#define identifiervalue
```

Example:

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

#define LENGTH 10
#define WIDTH  5
#define NEWLINE '\n'

int main() {
   int area;
```

```
    area = LENGTH * WIDTH;
    NSLog(@"The value of area : %d", area);
    NSLog(@"%c", NEWLINE);
    return 0;
}
```

const Keyword

Constants of a specific type can be declared with the const prefix, as seen below.

```
const typevariable = value;
```

Example:

```
#import <Foundation/Foundation.h>
int main() {
   const int  LENGTH = 20;
   const int  WIDTH  = 15;
   const char NEWLINE = '\n';
   int area;
   area = LENGTH * WIDTH;
   NSLog(@"The value of area : %d", area);
   NSLog(@"%c", NEWLINE);
   return 0;
}
```

OPERATORS IN OBJECTIVE-C

An operator in Objective-C is a symbol that instructs the compiler to do particular mathematical or logical operations. The Objective-C language has several built-in operators, including the following:

- Arithmetic Operators
- Assignment Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Misc Operators

This session will walk us through arithmetic, relational, logical, bitwise, assignment, and other operators.

ARITHMETIC OPERATORS IN OBJECTIVE-C

The table mentioned below lists all the arithmetic operators offered by the Objective-C programming language. Assume variable A has a value of 20, and variable B has a value of 30, then.

Operator	Description	Example
+	Adds the two operands	C + D will give 30
-	Subtracts the second operand from the first	C – D will give –12
*	Multiplies both the operands	C * D will give 100
/	Divides numerator by the denominator	D / C will give 3
%	Modulus Operator and the remainder of after integer division	D % C will give 0
++	The increment operator increases the integer value by one	C++ will give 14
––	The decrement operator decreases the integer value by one	C–– will give 7

RELATIONAL OPERATORS

The table below lists all the relational operations provided by the Objective-C programming language. Assume variable C has a value of 20 and variable D has a value of 30, then.

Operator	Description	Example
==	Compares the values of two operands; if they are equal, the condition becomes true.	(C == D) is not true.
!=	Compares the values of two operands; if the values are not equal, the condition evaluates to true.	(C != D) is true.
>	If value of the left operand is larger than the value of the right operand, the condition is determined to be true.	(C > D) is not true.
<	The condition is true if value of the left operand is smaller than the value of the right operand.	(C < D) is true.
>=	The condition is true if value of the left operand is larger than or equal to the value of the right operand.	(C >= D) is not true.
<=	The condition is true if value of the left operand is less than or equal to the value of the right operand.	(C <= D) is true.

LOGICAL OPERATORS IN OBJECTIVE-C

The table mentioned below lists all of the logical operators provided by the Objective-C programming language. Assuming variable C is 1 and variable D is 0, then.

Operator	Description	Example
&&	Defined as the Logical AND operator. If both operands are non-zero, the condition is satisfied.	(C && D) is false.
||	Defined as the Logical OR Operator. If any of the two operands are non-zero, the condition is satisfied.	(C || D) is true.
!	Defined as the Logical NOT Operator. In Objective-C to reverse the logical state of its operand, use. When a condition is true, the Logical NOT operator returns false.	!(C && D) is true.

BITWISE OPERATORS

The bitwise operator operates on bits and performs operations bit by bit. The truth tables for &, |, and are shown below.

c	d	c & d	c | d	c ^ d
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if C = 70; and D = 23; now, in binary format, they will be as follows:

```
C = 0011 1100
D = 0000 1101
-----------------
C&D = 0000 1100
C|D = 0011 1101
C^D = 0011 0001
~C = 1100 0011
```

The bitwise operations provided by Objective-C are given in the table below. Assume variable C has a value of 70 and variable D has a value of 23, then.

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(C & D) will give 12, which is 0000 1100
|	If bit exists in both operands, the binary OR operator copies it.	(C | D) will give 61, which is 0011 1101
^	If bit is set in one operand but not both, the binary XOR operator replicates it.	(C ^ D) will give 49, which is 0011 0001
~	Binary Ones Complement Operator is a unary operator that 'flipping' bits.	(~C) will give –61, which is 1100 0011 in 2's complement form.
<<	Left Shift Binary Operator: The left operand's value is shifted left by the number of bits given by the right operand.	C << 2 will give 240, which is 1111 0000
>>	Right Shift Binary Operator: The value of the left operand is shifted right by the number of bits given by the right operand.	C >> 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 right operands' values to the left operand.	Z = X + Y will assign value of X + Y into Z
+=	It adds the right operand to the left operand and assigns the resulting value to the left operand.	Z += X is equivalent to Z = Z + X
-=	It subtracts the right operand from the left operand and assigns the resulting value to the left operand.	Z -= X is equivalent to Z = Z – X
*=	Multiplication AND assignment, This operation multiplies the right operand by the left operand and assigns the product to the left operand.	Z *= X is equivalent to Z = Z * X
/=	Divide AND assignment divide the left operand by the right operand and assign the resulting value to the left operand.	Z /= X is equivalent to Z = Z / X
%=	AND assignment operator, modulus. It uses two operands to calculate the modulus and assigns the result to the left operand.	Z %= X is equivalent to Z = Z % X

(Continued)

Operator	Description	Example
<<=	The assignment AND shift-left operator.	Z <<= 2 is same as Z = Z << 2
>>=	The assignment AND right shift operator.	Z >>= 2 is same as Z = Z >> 2
&=	The AND bitwise assignment operator.	Z &= 2 is same as Z = Z & 2
^=	OR is a bitwise exclusive assignment and assignment operator.	Z ^= 2 is same as Z = Z ^ 2
|=	OR inclusive bitwise and assignment operator.	Z |= 2 is same as Z = Z | 2

MISC OPERATORS ↦ SIZEOF & TERNARY

Other essential operations provided by Objective-C language include size of and ? :

Operator	Description	Example
sizeof()	Returns the size of the variable.	sizeof(x), where a is an integer, will return 4.
&	Returns the address of a variable.	&x; will give the actual address of the variable.
*	Pointer to a variable.	*x; will pointer to a variable.
? :	The Conditional Expression	If the Condition is true ? Then value A : Otherwise value B.

OPERATORS' PRECEDENCE IN THE OBJECTIVE-C

Operator precedence controls how words in an expression are grouped. This influences the evaluation of an expression.

Certain operators have greater precedence than others; for instance, the multiplication operator has higher precedence than the addition operator. For example, c = 17 + 3 * 4; here, c is assigned 29, not 80, because operator * has higher precedence than +, so it first gets multiplied with 3*4 and then added into 12.

Here, operators with the most significant precedence are positioned at the top of the table, while those with the lowest precedence are positioned at the bottom. Within an expression, operators with higher precedence will be evaluated first.

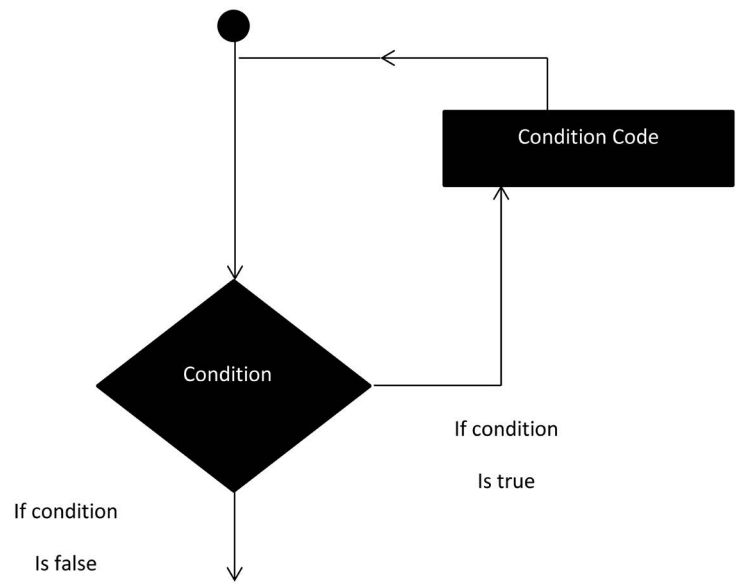
Category	Operator	Associativity
Postfix	() [] –>. ++ – –	Left to right
Unary	+ – ! ~ ++ – – (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right

(Continued)

Category	Operator	Associativity
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

LOOPS IN OBJECTIVE-C

There may be events when we need to execute a code block numerous times. In general, statements are performed in the following order: the first statement in a function is executed first, then the second, and so on. Control structures in programming languages allow for more sophisticated execution routes. A loop statement allows us to run a statement or collection of statements many times. The Objective-C programming language includes the following loop types to address looping needs.



Loop in Objective-C.

Sr. No.	Loop Type and Description
1	while loop While a specific condition is true, a sentence or set of statements is repeated. Before running the loop body, it checks the condition.
2	for loop The code that maintains the loop variable is abbreviated when a series of statements is executed many times.
3	do...while loop It's similar to a while statement. Only it checks the condition after the loop body.
4	nested loops A while, for, or do while loop can include one or more loops.

CONTROL STATEMENTS FOR LOOPS

Loop control statements alter the execution sequence. All automated objects produced in that scope are deleted when execution exits that scope.

Objective-C supports the following control statements. To learn more about the control statements, click the links below.

Sr. No.	Control Statement and Description
1	break statement The loop or switch statement is terminated, and execution is transferred to the statement immediately after the loop or switch.
2	continue statement The loop will skip the rest of its body and instantly retest its state before repetition.

INFINITE LOOP

If condition never becomes false, loop becomes endless. Traditionally, the for loop is used for this purpose. Because none of the three for loop expressions are necessary, we may create an infinite loop by leaving the conditional expression empty.

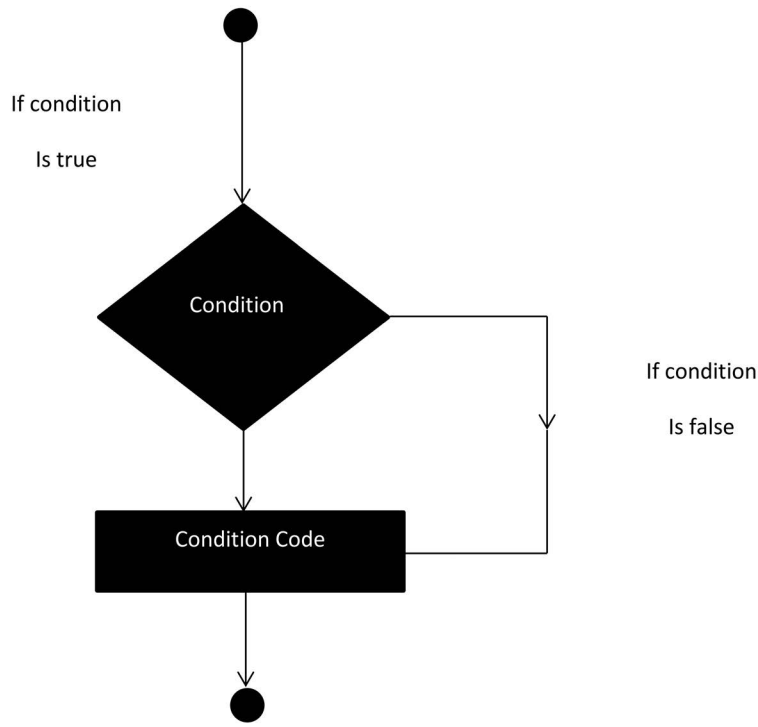
```
#import <Foundation/Foundation.h>

int main () {

   for( ; ; ) {
      NSLog(@"loop will run forever.\n");
   }

   return 0;
}
```

It is believed to be true when the conditional statement is missing. Although we may use an initialization and increment expression, Objective-C programmers prefer to use them for(;;) construct to represent an endless loop.

DECISION MAKING IN OBJECTIVE-C

The programmer must define one or more conditions to be evaluated or tested by the program, a statement or statements to be performed if the condition is discovered to be true, and optionally, further statements to be run if the condition is decided to be false.

The general shape of a common decision-making framework in most programming languages is shown below.



Decision making in Objective-C.

The Objective-C programming language treats any non-zero and non-null value as true, but any zero or null value is false.

The Objective-C programming language has the following decision-making statements.

Sr. No.	Statement and Description
1	if statement A boolean expression is followed by one or more statements in an if statement.
2	if...else statement When boolean expression is false, the if an optional else statement follows the statement.
3	nested if statements One if or else if statement can use inside another if or else if statement(s).
4	switch statement A switch statement checks a variable for equality against a set of values.
5	nested switch statements One switch statement can be used inside another switch statement(s).

THE ? : OPERATOR

We discussed the conditional operator?:, which may be used to replace if… else expressions. It takes the overall shape shown below.

```
Exp1?  Exp2 : Exp3;
```

Where Exp1, Exp2, and Exp3 are expressions. Take note of the colon's use and location.

A? expression's value is decided as follows: Exp1 is evaluated. If it is true, Exp2 is evaluated and the value of the entire? Expression is determined. If Exp1 is false, Exp3 is evaluated, and its value becomes the expression's value.

FUNCTIONS IN OBJECTIVE-C

A function in Objective-C is a collection of statements that execute a particular activity. Every Objective-C program has one C function, called main(), and even the simplest applications may declare extra functions.

We may break up your code into various functions. It is up to us how we split our code across distinct functions, but logically, each function should fulfil a particular task.

A function declaration informs the compiler about the function's name, return type, and arguments. The body of a function is specified by its definition.

In Objective-C, the function is referred to as a method.

The Objective-C base framework contains various built-in methods that our application may call. For instance, the appendString() function appends one string to another.

A method is recognized by several names, including function, subroutine, process, etc.

CREATING A METHOD

The following is the general form of a method definition in the Objective-C programming language:

```
- (returntype) methodname:( argumentType1 )
argumentName1
joiningArgument2:( argumentType2 )argumentName2 ....
joiningArgumentn:( argumentTypen )argumentNamen {
   body of function
}
```

A method specification consists of a method header and a method body in the Objective-C programming language. Here are all of the components of a method:

- Return type: In Objective-C a method may return a value as its returntype. The function's data type of value is specified by return type. Some methods carry out the necessary tasks but do not return a value. The term void is used as the returntype in this scenario.
- Method name: This is the method's actual name. The method signature comprises the method name and the argument list.
- Arguments: An argument functions similarly to a placeholder. When we call a function, we pass an argument with a value. This value is known as the actual parameter or argument. A method's parameter list specifies the type, order, and the number of parameters. Arguments are optional; a method may not have any arguments.
- Joining argument: A joining argument makes it easier to read and more explicit while invoking it.
- Method body: The method body comprises statements that describe what the method performs.

Example: The following is the source code for the max() method. This function accepts two parameters, numb1 and numb2, and returns the greater of the two:

```
/* function returning max between two numbers */
- (int) max:(int) numb1 secondNumber:(int) numb2 {
```

```
    /* the local variable declaration */
    int result;

    if (numb1 > numb2) {
       result = numb1;
    } else {
       result = numb2;
    }

    return result;
}
```

DECLARATIONS OF METHOD

A method declaration gives the compiler the function's name and how to invoke it. The function's actual body in Objective-C method can be specified independently.

A method declaration in Objective-C consists of the following components:

```
- (returntype) functionname:( argumentType1 )
argumentName1
joiningArgument2:( argumentType2 )argumentName2 ....
joiningArgumentn:( argumentTypen )argumentNamen;
```

The method declaration for the above-described function max() is as follows:

```
-(int) max:(int)numb1 andNum2:(int)numb2;
```

When defining a method in one source file and calling it in another, method declaration is necessary. In this scenario, the function should be declared at the start of the file invoking the function.

CALLING A METHOD

When creating an Objective-C method, we define what the function must accomplish. We must invoke that function to complete the specified operation to utilize a method.

When a program invokes a function, program control is passed to the invoked method. A called method performs a stated task and returns

program control to the main program when its return statement or function-ending closing brace is reached.

To call a method, simply give the needed arguments along with the method name, and if the method returns a result, we may save it. As an example,

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

@interface SampleClass:NSObject
/* method-declaration */
- (int)max:(int)numb1 andNumb2:(int)numb2;
@end

@implementation SampleClass

/* method returning max between two numbers */
- (int)max:(int)numb1 andNumb2:(int)numb2 {

   /* the local variable declaration */
   int result;

   if (numb1 > numb2) {
      result = numb1;
   } else {
      result = numb2;
   }

   return result;
}

@end

int main () {

   /* the local variable definition */
   int x = 200;
   int y = 100;
   int ret;

   SampleClass *sampleClass = [[SampleClass alloc]
init];
```

```
   /* calling method to get the max value */
   ret = [sampleClass max:x andNumb2:y];

   NSLog(@"Max value is : %d\n", ret );
   return 0;
}
```

FUNCTION ARGUMENTS

If a function accepts arguments, it must define variables that accept the arguments' values in Objective-C. In Objective-C, these variables are known as the function's formal parameters.

The formal parameters operate similarly to other local variables within the function, generated upon entrance and removed upon departure.

There are two methods for passing arguments to a function when calling it.

Sr. No.	Call Type and Description
1	Call by value This method replicates the real value of an argument into the function's formal parameter. Changes to the parameter within the function do not affect the argument in this case.
2	Call by reference This method inserts an argument's address into the formal parameter. The address is utilized within the function to obtain the actual parameter used in the call. This signifies that changes to the parameter affect the argument.

To pass arguments, Objective-C employs call by value by default. In general, this implies that code within a function cannot change the arguments used to call the function, and the preceding example is utilized the same way when using the max() function.

BLOCKS IN OBJECTIVE-C

An Objective-C class is a type of object that combines data with associated functionality. It is sometimes more appropriate to express a single job or unit of action rather than the collection of methods.

The addition of blocks to C, Objective-C, and C++ has made it possible to create separate chunks of code that can be handed to methods and functions like values. Add blocks to collections like NSArray or NSDictionary as they are Objective-C objects. They are comparable to closures or lambdas in other programming languages since they may also catch values from the surrounding scope.

Simple Block Declaration Syntax

```
returntype (^block_Name)(argument_Type);
```

Implementation of a Simple Block

```
returntype (^block_Name)(argument_Type)= ^{
};
```

Here's an easy example:

```
void (^simpleBlock)(void) = ^{
    NSLog(@"This is block");
 };
```

BLOCKS TAKE ARGUMENTS AND RETURN VALUES

Blocks, like methods and functions, can take arguments and return values.

Here's a simple example of creating and running a block with parameters and return values.

```
double (^multiplyTwoValues)(double, double) =
   ^(double firstValue, double secondValue) {
      return firstValue * secondValue;
   };

double result = multiplyTwoValues(12,41);
NSLog(@"Result is %f", result);
```

BLOCKS USING THE TYPE DEFINITIONS

Here's an easy example of using typedef in a block. Please remember that this sample does not yet operate with the online compiler. To execute the same, use XCode.

```
#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() {

   /* my first program in the Objective-C */
   SampleClass *sampleClass = [[SampleClass alloc]
init];
   [sampleClass performActionWithCompletion:^{
      NSLog(@"The Completion is called to intimate
action is performed");
   }];

   return 0;
}
```

NUMBERS IN OBJECTIVE-C

To preserve fundamental data types such as int, float, and bool in object form in the Objective-C programming language.

The following table lists the most significant Objective-C methods for dealing with NSNumber.

Sr. No.	Method and Description
1	+ (NSNumber *)numberWithBool:(BOOL)value Creates and returns an NSNumber object with the provided value as a BOOL.
2	+ (NSNumber *)numberWithChar:(char)value Creates and returns an NSNumber object with the specified value as a signed char.
3	+ (NSNumber *)numberWithDouble:(double)value Creates and returns an NSNumber object with the specified value as a double.
4	+ (NSNumber *)numberWithFloat:(float)value Creates and returns an NSNumber object with the specified value as a float.

(Continued)

Sr. No.	Method and Description
5	+ (NSNumber *)numberWithInt:(int)value Creates and returns an NSNumber object with the specified value as a signed int.
6	+ (NSNumber *)numberWithInteger:(NSInteger)value Creates and returns an NSNumber object with the supplied value, treated as an NSInteger.
7	– (BOOL)boolValue Returns receiver's value as a BOOL.
8	– (char)charValue Returns receiver's value as a char.
9	– (double)doubleValue Returns receiver's value as a double.
10	– (float)floatValue Returns receiver's value as a float.
11	– (NSInteger)integerValue Returns receiver's value as an 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 basic example of using NSNumber to multiply two numbers and return the result.

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

@interface SampleClass:NSObject
- (NSNumb *)multiplyA:(NSNumb *)a withB:(NSNumb *)b;
@end

@implementation SampleClass

- (NSNumb *)multiplyA:(NSNumb *)a withB:(NSNumb *)b {
   float numb1 = [a floatValue];
   float numb2 = [b floatValue];
   float product = numb1 * numb2;
   NSNumb *result = [NSNumb numbWithFloat:product];
   return result;
}

@end

int main() {
```

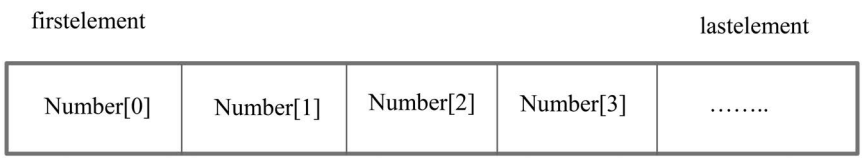
```
   NSAutoreleasePool * pool = [[NSAutoreleasePool
alloc] init];

   SampleClass *sampleClass = [[SampleClass alloc]
init];
   NSNumb *a = [NSNumb numbWithFloat:10.5];
   NSNumb *b = [NSNumb numbWithFloat:10.0];
   NSNumb *result = [sampleClass multiplyA:a withB:b];
   NSString *resultString = [result stringValue];
   NSLog(@"Product is %@",resultString);

   [pool drain];
   return 0;
}
```

ARRAYS IN OBJECTIVE-C

The array data structure in the Objective-C programming language may hold a fixed-size sequential collection of the elements of the same type. A collection of data is stored in an array, although it is generally more convenient to conceive an array as a collection of variables of the same type.

Instead of defining individual variables like number0, number1,…, and number99, we define one array variable like numbers and use numbers[0], numbers[1,…, and numbers[99] to represent individual variables. An index is used to access a specific element in an array.

All arrays are made up of contiguous memory locations. The first element refers to the lowest address, while the last element corresponds to the highest address.



Array in Objective-C.

DECLARING ARRAYS

In Objective-C, a programmer defines an array by describing the kind of elements and the quantity of elements needed by the array as follows:

```
type array_Name [ array_Size ];
```

This is referred to as a single-dimensional array. The arraySize constant must be bigger than zero, and the type can be any acceptable Objective-C data type. For example, to declare a 10-element array of type double named balance, use the following statement:

```
double balance[20];
```

Balance is now a variable array that can store upto ten double numbers.

ARRAYS INITIALIZATION

In Objective-C, we may initialize an array one by one or with a single line, as seen below.

```
double balance[5] = {2000.0, 3.0, 4.4, 19.0, 56.0};
```

The number of values between the braces { } cannot exceed the number of items declared for the array between the square brackets [ ]. The following is an example of assigning a single array element.

If the array size is not specified, an array only large enough to accommodate the initialization is generated. As a result, if we write:

```
double balance[] = {2000.0, 3.0, 4.4, 19.0, 56.0};
```

We will construct the same array as in the previous example.

```
balance[4] = 56.0;
```

The preceding statement assigns a value of 56.0 to the array's fifth member. Because all arrays have 0 as the index of their first element, also known as the base index, the array with the fourth index will be the fifth, that is, the last element.

ACCESSING ARRAY ELEMENTS

The array name is indexed to find an element. This is accomplished by following the array name with the element's index enclosed in square brackets. As an example,

```
double salary = balance[8];
```

The above code will choose the 9th member from the array and assign its value to the salary variable. The following is an example that employs all three concepts mentioned earlier: declaration, assignment, and array access.

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

int main () {
   int nm[ 10 ];   /* nm is an array of 10 integers */
   int x,y;

   /* initialize elements of array nm to 0 */
   for ( x = 0; x < 10; x++ ) {
      nm[ x ] = x + 100;    /* set element at location
x to x + 100 */
   }

   /* output each array element's value */
   for (y = 0; y < 10; y++ ) {
      NSLog(@"The Element[%d] = %d\n", y, nm[y] );
   }

   return 0;
}
```

ARRAYS IN OBJECTIVE-C IN DEPTH

Arrays are vital in Objective-C and require a lot more information. The following are a few key array ideas that any Objective-C developer should understand:

Sr. No.	Concept and Description
1	Multi-dimensional arrays Objective-C supports multi-dimensional arrays. The two-dimensional array is the most basic type of multi-dimensional array.
2	Passing arrays to functions We can pass a pointer to an array to the method by supplying the array's name without an index.
3	Return array from a function A function in Objective-C can return an array.
4	Pointer to an array By merely specifying the array name without any index, we may produce a reference to the first element of an array.

POINTERS IN OBJECTIVE-C

Objective-C pointers are simple and enjoyable to learn. Some Objective-C programming tasks are easier to accomplish with pointers, while others, such as dynamic memory allocation, cannot be performed without them. As a result, learning pointers are required to become a proficient Objective-C programmers. Let us begin by studying them in small and easy steps.

As you may know, each variable is a memory location. Each memory location has an address in Objective-C that can be accessed using the ampersand (&) operator, representing a memory address. Consider the following example, which prints the addresses of the variables specified.

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

int main () {
   int  var1;
   char var2[10];

   NSLog(@"The Address of var1 variable is: %x\n",
&var1 );
   NSLog(@"The Address of 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 address of another variable, i.e., the memory location's direct address. Like any variable or constant, a pointer must be declared before using them to hold any variable address. A pointer variable declaration has the following general form:

```
type *var_name;
```

In Objective-C the pointer's base type is type, which must be a valid Objective-C data type, and the pointer variable's name is var_name. The asterisk * in Objective-C is used to declare a pointer is the same asterisk

used for multiplication. In this case, though, the asterisk indicates a variable as a pointer. The valid pointer declarations are as follows:

```
int    *ip;    /* pointer to integer */
double *dp;    /* pointer to double */
float  *fp;    /* pointer to float */
char   *ch     /* pointer to character */
```

The true data type of all pointer values, whether integer, float, character or otherwise, is a lengthy hexadecimal number representing a memory location. The sole distinction between pointers of various data types is the data type of the variable or constant pointed to by the pointer.

How Do Pointers Work?

There are just a handful of significant actions that we will do regularly using pointers. (a) we create a pointer variable, (b) we assign the address of a variable to a pointer, and (c) we ultimately access the value at the address accessible via the pointer variable. This is accomplished using the unary operator *, which returns the variable's value at the location given by its argument. The following illustration employs these operations:

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

int main () {
   int  var = 30;    /* actual variable-declaration */
   int  *ip;         /* pointer variable-declaration */

   ip = &var;       /* store address of var in the
pointer variable*/

   NSLog(@"Address of var variable: %x\n", &var  );

   /* address stored in the pointer variable */
   NSLog(@"Address stored in the ip variable: %x\n",
ip );

   /* access value using the pointer */
   NSLog(@"The Value of *ip variable: %d\n", *ip );

   return 0;
}
```

OBJECTIVE-C NULL POINTERS

In Objective-C, it is usually a good idea to assign a NULL value to a pointer variable if the allocated address is unknown. This is done during variable declaration. A null pointer is a pointer in Objective-C that has been assigned the value NULL.

The NULL pointer in Objective-C is a zero-valued constant declared in numerous standard libraries. Take a look at the following program:

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

int main () {
   int  *ptr = NULL;
   NSLog(@"Value of the ptr is : %x\n", ptr  );
   return 0;
}
```

The following outcome is produced when the given code is compiled and executed in Objective-C:

```
Value of ptr is: 0
```

Most operating systems prohibit applications from accessing memory at address 0 because the operating system reserves that memory. However, memory address 0 is significant since it indicates that the pointer is not meant to point to an accessible memory region. However, if a pointer has the null (zero) value, it is presumed to point to nothing.

To check for a null pointer, use the following if statement:

```
if(ptr)     /* succeeds if x is not null */
if(!ptr)    /* succeeds if x is null */
```

DETAILS ABOUT OBJECTIVE-C POINTERS

Pointers are highly crucial in Objective-C programming and have several yet simple ideas. The following are a few key pointer concepts that any Objective-C developer should understand:

Sr. No.	Concept and Description
1	Objective-C – Pointer arithmetic On pointers, four arithmetic operators are available: ++, –-, +, –
2	Objective-C – Array of pointers Arrays may be used to contain a collection of pointers.

(Continued)

Sr. No.	Concept and Description
3	Objective-C – Pointer to pointer Objective-C supports pointer on the pointer and so on.
4	Passing pointers to functions in Objective-C Passing an argument by reference or address allows the called function to alter the provided argument in the calling code.
5	Return pointer from functions in Objective-C A function in Objective-C can return a pointer to a local variable, a static variable, or dynamically allocated memory.

STRINGS IN OBJECTIVE-C

The string is represented in the Objective-C programming language by NSString, and its subclass NSMutableString provides numerous methods for constructing string objects. The Objective-C @"…" syntax is the easiest way to generate a string object.

```
NSString *greeting = @"Heyyy";
```

The following is a simple example of producing and printing a string.

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

int main () {
   NSString *greeting = @"Heyyy";
   NSLog(@"The Greeting message is: %@\n", greet );

   return 0;
}
```

Objective-C provides a plethora of techniques for manipulating strings.

Sr. No.	Method and Purpose
1	– (NSString *)capitalizedString; Returns the 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.

(Continued)

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 created by utilizing a given format string as a template and then substituting the remaining argument values.
8	– (NSInteger)integerValue; The NSInteger value of the receiver's text is returned.
9	– (BOOL)isEqualToString:(NSString *)aString; Returns a Boolean value indicating if a provided string and the receiver are equal using a literal Unicode-based comparison.
10	– (NSUInteger)length; The number of Unicode characters in the receiver is returned.
11	– (NSString *)lowercaseString; The lowercased representation of the receiver is returned.
12	– (NSRange)rangeOfString:(NSString *)aString; The range of the first occurrence of a specified string within the receiver is returned.
13	– (NSString *)stringByAppendingFormat:(NSString *)format ...; Returns a string created by attaching a string formed from a provided format string and the following parameters to the receiver.
14	– (NSString *)stringByTrimmingCharactersInSet:(NSCharacterSet *)set; Returns a new string created by eliminating characters from both ends of the receiver part of a specific character set.
15	– (NSString *)substringFromIndex:(NSUInteger)anIndex; Returns a new string containing the receiver's characters from the one at the provided index to the end.

The following example employs a few of the functions discussed above:

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

int main () {
   NSString *str1 = @"Hello";
   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 the str3 after 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 enable us to construct types of variables that may store many data items of the same kind. Still, the structure is another user-defined data type accessible in Objective-C programming that allows us to mix data items of various types.

Structures are used to represent data. Assume we wish to keep track of your library books. We might wish to keep note of the following characteristics of each book:

- Title
- Author
- Subject
- Book ID

CREATING A STRUCTURE

The struct statement is required to define a structure. The struct statement creates a new data type for our code with more than one member. The struct statement format is illustrated below.

```
struct [structure tag] {
    member-definition;
    member-definition;
    ...
    Member-definition;
 } [one or more structure variables];
```

The structure tag is optional in Objective-C, and each member definition is a standard variable definition, such as int i; float f; or any other acceptable variable definition. Although this is optional, we can declare one or more structure variables after the structure's definition, before the final semicolon. This is how we would declare 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 gain access to any structure member (.). The member access operator in Objective-C is represented by a period between the name of the structure variable and the name of the structure member that we want to access. To define variables of the structure type, we would use the struct keyword. The example below demonstrates 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;

   /* print Book1 info */
   NSLog(@"Book 1 title : %@\n", Book1.title);
   NSLog(@"Book 1 authors : %@\n", Book1.authors);
   NSLog(@"Book 1 subjects : %@\n", Book1.subjects);
   NSLog(@"Book 1 bookid : %d\n", Book1.bookid);

   /* print Book2 info */
   NSLog(@"Book 2 title : %@\n", Book2.title);
   NSLog(@"Book 2 authors : %@\n", Book2.authors);
   NSLog(@"Book 2 subjects : %@\n", Book2.subjects);
   NSLog(@"Book 2 bookid : %d\n", Book2.bookid);

   return 0;
}
```

FUNCTION ARGUMENTS AS STRUCTURES

A structure can be sent as a function parameter as any other variable or pointer can. We would access structural variables in the same way we did in the preceding example.

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

struct Books {
```

```
   NSString *title;
   NSString *authors;
   NSString *subjects;
   int   bookid;
};

@interface SampleClass:NSObject
/* function-declaration */
- (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 = @" Tutorial  of Objective-C
Programming";
   Book1.bookid = 4387307;

   /* book2 specification */
   Book2.title = @"Telecom Billing";
   Book2.author = @"Sara Ali";
   Book2.subject = @" Tutorial of Telecom Billing";
   Book2.book_id = 9875701;

   SampleClass *sampleClass = [[SampleClass alloc]
init];
```

```
   /* print Book1 info */
   [sampleClass printBook: Book1];

   /* Print Book2 info */
   [sampleClass printBook: Book2];

   return 0;
}
```

POINTERS TO STRUCTURES

As seen below, we may define pointers to structures in the same way that define pointers to any other variable.

```
struct Books *struct-pointer;
```

The address of a structural variable can now be stored in the above-described pointer variable in Objective-C. In Objective-C to find the address of a structure variable, use the & operator before the structure's name, as shown below.

```
Struct-pointer = &Book1;
```

To access the structure members from a pointer to that structure, use the -> operator as shown below.

```
Struct-pointer->title;
```

Let us rewrite the above example using a structure pointer; perhaps, this will help us comprehend the notion.

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

struct Books {
   NSString *title;
   NSString *authors;
   NSString *subjects;
   int   book-id;
};

@interface SampleClass:NSObject
/* function-declaration */
```

```
- (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 */

   /* book 1 specification */
   Book1.title = @"Objective-C Programming";
   Book1.authors = @"Luka Mli";
   Book1.subjects = @" Tutorial of Objective-C
Programming";
   Book1.bookid = 4387307;

   /* book 2 specification */
   Book2.title = @"Telecom Billing";
   Book2.authors = @"Sara Ali";
   Book2.subjects = @" Tutorial 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

Bit Fields allow data to be packed into a structure. This is especially helpful when memory or data storage is limited.

Packing several objects into a machine word, such as, 1 bit flags can be compacted.

External file formats can be read in if they are not standard. As an example, consider 9-bit integers.

We may achieve this in an Objective-C structure declaration by putting: bit length after the variable. As an example,

```
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;
```

The packed_struct, in this case, has six members: Four 1-bit flags f1..f3, a4 bit type, and a 9-bit my_int

The bit mentioned above fields are automatically packed as compactly as feasible by Objective-C, provided the field's maximum length is less than or equal to the computer's integer word length. If this is not the case, specific compilers may allow for field memory overlap, while others may store the next field in the following word.

PREPROCESSORS IN OBJECTIVE-C

The Objective-C Preprocessor is a distinct step in the compilation process, not part of the compiler. An Objective-C Preprocessor is just a text replacement tool that informs the compiler to perform the necessary preprocessing before actual compilation. The Objective-C Preprocessor will be referred to as the OCPP.

A pound sign (#) precedes all preprocessor commands. It must be the first non-blank character, and a preprocessor directive should begin in the first column for readability. The section that follows lists all of the key preprocessor directives.

Sr. No.	Directive and Description
1	#define Substitutes preprocessor macro
2	#include Inserts particular header from another file.
3	#undef Undefines preprocessor macro
4	#ifdef If this macro is defined, it returns true.
5	#ifndef If this macro is not defined, it returns true.
6	#if Checks if a compile-time condition is true.
7	#else Alternative for #if
8	#elif #else an #if in the one statement.
9	#endif Ends the preprocessor conditional.
10	#error Prints the error message on stderr.
11	#pragma Issues special commands to the compiler using standardized method.

EXAMPLES OF PREPROCESSORS

Examine the samples below to understand various directives better.

```
#define MAXARRAY_LENGTH 22
```

This directive instructs the OCPP to replace MAXARRAY_LENGTH instances with 20. To improve readability, use #define for constants.

```
#import <Foundation/Foundation.h>
#include "myheader.h"
```

These directives instruct the OCPP to retrieve foundation.h from the Foundation Framework and insert it into the current source file. The next line instructs OCPP to retrieve myheader.h from the local location and insert it into the current source file.

```
#undef  FILE_SIZE
#define FILE_SIZE 41
```

This instructs the OCPP to undefine the current FILE_SIZE to 41.

```
#ifndef MESSAGE
   #define MESSAGE "We wish"
#endif
```

This instructs the OCPP to define MESSAGE only if it has not already been defined.

```
#ifdef DEBUG
/* Our debugging statements here.. */
#endif
```

If DEBUG is defined, this instructs the OCPP to process the statements contained. This is essential if the -DDEBUG option is passed to the gcc compiler during compilation. This defines DEBUG, allowing us to toggle debugging on and off during compilation.

PREDEFINED MACROS

Several macros are defined in ANSI C. Although each is accessible for use in programming, the predefined macros are not modified directly.

Sr. No.	Macro and Description
1	__DATE__ The current date as character literal in the "MMM DD YYYY" format
2	__TIME__ The current time as character literal in the "HH:MM:SS" format
3	__FILE__ This contains current filename as string literal.
4	__LINE__ This contains the current line number as decimal constant.
5	__STDC__ Defined as 1 when the compiler complies with ANSI standard.

Consider the following example:

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

int main() {
   NSLog(@"The File :%s\n", __FILE__ );
   NSLog(@"The Date :%s\n", __DATE__ );
```

```
    NSLog(@"The Time :%s\n", __TIME__ );
    NSLog(@"The Line :%d\n", __LINE__ );
    NSLog(@"ANSI :%d\n", __STDC__ );

    return 0;
}
```

OPERATORS OF PREPROCESSORS

To assist us in constructing macros, the Objective-C preprocessor provides the following operators.

Macro Continuation (\)

A macro must typically be confined to a single line. Use the macro continuation operator in Objective-C to continue a macro that is too long for a single line. As an example,

```
#define  message_for(x, y)  \
    NSLog(@#x " and " #y ": We miss you\n")
```

Stringize (#)

The stringize or number-sign operator ("#") transforms a macro argument into a string constant when used within a macro definition. Use this operator in a macro with a predefined argument or parameter list. As an example,

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

#define  message_for(x, y)  \
    NSLog(@#x " and " #y ": We miss you\n")

int main(void) {
    message_for(Carole, Debra);
    return 0;
}
```

Token Pasting (##)

The token-pasting operator (##) joins two parameters within a macro declaration. It enables the joining of two different tokens in the macro specification into a single token. As an example,

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

```
#define tokenpaster(n) NSLog (@"token" #n " = %d",
token##n)

int main(void) {
   int token24 = 60;

   tokenpaster(24);
   return 0;
}
```

defined() Operator

The preprocessor defined operator is used in the constant expressions to verify if an identifier is defined using #define. The value is true if the provided identifier is defined (non-zero). The value is false if the symbol is not specified (zero). The following is the definition of the defined operator:

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

#if !defined (MESSAGE)
   #define MESSAGE "We wish"
#endif

int main(void) {
   NSLog(@"Here message is: %s\n", MESSAGE);
   return 0;
}
```

PARAMETERIZED MACROS

Simulating functions using parameterized macros is one of OCPP's most powerful features. For example, we may have the following code to square a number:

```
int square(int a) {
    return a * a;
 }
```

We may modify the preceding code using a macro as follows:

```
#define square(a) ((a) * (a))
```

Before using them, macros with parameters must be specified with the #define directive. The parameter list enclosed by parentheses must come

immediately after the macro name. Spaces are not permitted between the macro name and the open parenthesis. As an example,

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

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

int main(void) {
   NSLog(@"The Max between 30 and 20 is %d\n", MAX(20,
30));
   return 0;
}
```

Typedef IN OBJECTIVE-C

The Objective-C programming language includes a typedef keyword used to rename a type. Below mentioned is an example of how to define the term BYTE for one-byte numbers:

```
typedef unsigned char BYTE;
```

Following this type definition, the identifier BYTE can use as an abbreviation for the type unsigned char, for instance:

```
BYTE  by1, by2;
```

Uppercase letters are used by convention for these definitions to remind the user that the type name is only a symbolic abbreviation. However, lowercase letters can also be used, as seen below.

```
typedef unsigned char byte;
```

We may also use typedef to assign a name to a user-defined data type. For example, we may use typedef with the structure to build a new data type and then use that data type to declare structure variables, as seen below explicitly.

```
#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 = @"TutorialsPoint";
   book.subjects = @"Programming-tutorial";
   book.bookid = 200;

   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 is an Objective-C directive that is used to declare aliases for various data types, similar to typedef but with the following differences:

Typedef can only give symbolic names to types, but #define may also be used to define aliases for values, such as 1 as ONE, etc.

The compiler interprets typedef declarations, whereas the pre-processor processes #define statements.

The following is the most basic use of #define.

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

#define TRUE  1
#define FALSE 0

int main( ) {
   NSLog( @"The Value of TRUE : %d\n", TRUE);
   NSLog( @"the Value of FALSE : %d\n", FALSE);

   return 0;
}
```

TYPE CASTING IN OBJECTIVE-C

Type casting is a method of converting a variable from one data type to another. For example, if we want to convert a long number to a basic integer, we may use the type cast long to int. As seen below, we may explicitly use the cast operator to change values from one type to another.

```
(typename) expression
```

In Objective-C, we often use CGFloat for floating-point operations, derived from the fundamental type of float in 32-bit cases and double in 64-bit cases. Consider the following example, in which the cast operator divides one integer variable by another in a floating-point operation:

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

int main() {
   int sum = 27, count = 15;
   CGFloat mean;

   mean = (CGFloat) sum / count;
   NSLog(@"The Value of mean : %f\n", mean );

   return 0;
}
```

Note that the cast operator takes precedence over the division operator. Thus, the sum value is first changed to type double before being divided by count, returning a double value.

Type conversions can be implicit (done automatically by the compiler) or explicit (expressed explicitly using the cast operator). When type conversions are required, it is considered good programming practice to utilize the cast operator.

INTEGER PROMOTION

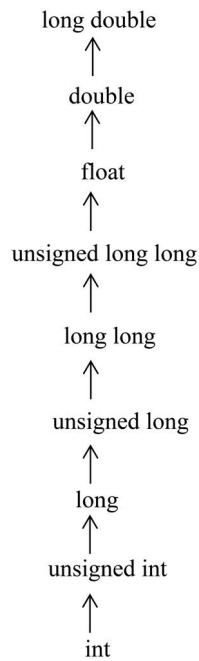
Integer promotion is when values of the integer type "smaller" than int or unsigned int are upgraded to int or unsigned int. Consider adding a character to an int.

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

int main() {
```

```
    int  x = 27;
    char c = 'c';
    int sum;

    sum = x + c;
    NSLog(@"The Value of sum : %d\n", sum );

    return 0;
}
```

USUAL ARITHMETIC CONVERSION

The usual arithmetic conversions are done implicitly to cast their values in a common type. If operands continue to be of various types, the compiler converts them to the type that appears highest in the following hierarchy.



Usual arithmetic conversions.

The assignment operators and the logical operators && and || do not undergo the standard arithmetic conversions. To further comprehend the concept, consider the following example.

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

int main() {
   int  x = 27;
   char c = 'c';
   CGFloat sum;

   sum = x + c;
   NSLog(@"The Value of sum : %f\n", sum );
   return 0;
}
```

LOG HANDLING IN OBJECTIVE-C

In this section, we will describe log handling and provide appropriate examples.

NSLog METHOD

We utilize the NSLog function in the Objective-C programming language to print logs, which we first used in the Hello World example.

Consider a simple code that prints the words "Hello Everyone."

DISABLING LOGS IN THE LIVE Apps

Because we utilize NSLogs in our application, it will be written in device logs, which is not good in a live build. As a result, we employ a type definition for printing logs, as illustrated below.

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

#if DEBUG == 0
#define DebugLog(...)
#elif DEBUG == 1
#define DebugLog(...) NSLog(__VA_ARGS__)
#endif

int main() {
   DebugLog(@"Debug log, our custom addition gets \
   printed during the debug only" );
   NSLog(@"The NSLog gets printed always" );
   return 0;
}
```

ERROR HANDLING IN OBJECTIVE-C

Error handling in Objective-C programming is handled via the Foundation framework's NSError class.

An NSError object has more detailed and extensible error information than an error code or error text. An NSError object has three fundamental attributes: an error domain (expressed by a string), a domain-specific error code, and a user info dictionary providing application-specific information.

NSError

NSError objects are used in Objective-C programs to provide information about runtime faults that users should be aware of. Typically, a program will display this error information in a dialog or sheet. However, it may interpret data and either request that the user attempt to recover from the error or seek to repair the issue on its own.

The NSError Object consists of:

- Domain: The error domain must not be nil and can be one of the preset NSError domains or an arbitrary string specifying a new domain.
- Code: The error code.
- User Info: The error's userInfo dictionary, which may be null.

The example below demonstrates how to generate a custom error.

```
NSString *domain = @"com.MyCompany.MyApplication.
ErrorDomain";
NSString *desc = NSLocalizedString(@"Unable to
complete process", @"");
NSDictionary *userInfo = @{ NSLocalizedDescriptionKey
: desc };
NSError *error = [NSError errorWithDomain:domain
code:-101 userInfo:userInfo];
```

Here is the complete code for the error as mentioned in the 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 Test-Name";
   } else {
      NSString *domain = @"com.MyCompany.
MyApplication.ErrorDomain";
      NSString *desc =@"Unable to complete process";
      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(@"Error-finding Name1: %@",error);
   } else {
      NSLog(@"Name1: %@",name1);
   }

   error = nil;
```

```
   NSString *name2 = [sampleClass
getEmployeeNameForID:2 withError:&error];

   if(error) {
      NSLog(@"Error-finding Name2: %@",error);
   } else {
      NSLog(@"Name2: %@",name2);
   }

   [pool drain];
   return 0;
}
```

COMMAND-LINE ARGUMENTS

When your Objective-C programs are executed, we can provide specific values from the command line to them. These data are known as command-line arguments, and they are frequently helpful for your program, particularly when we want to control our program from outside rather than hard-coding those values inside the code.

The command-line arguments are handled using the main() function parameters, where argc is the number of arguments supplied, and argv[] is a pointer array pointing to each argument passed to the program. The following is a simple example that checks for command-line arguments and takes appropriate 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 too many arguments.\n");
   } else {
      NSLog(@"One argument expected.\n");
   }
}
```

When the preceding code is built and performed with a single parameter, say "testing," the outcome is as follows:

```
The argument supplied is testing.
```

The following result is obtained when the below code is built and performed with two parameters, say testing1 and testing2.

Supplied too many arguments.

The following result is obtained when the code is built and executed without any arguments.

```
One argument is expected.
```

It should be noted that argv[0] contains the program's name, argv[1] is a reference to the first command-line parameter provided, and *argv[n] is the last argument. If no arguments are given, argc is set to one; otherwise, argc is set at 3 if one argument is supplied.

A space separates all command line parameters. However, if the argument contains a space, you can pass it by enclosing it in double quotes "" or single quotes ''. Let us rewrite the above example, this time printing the program name and passing 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 too many arguments.\n");
   } else {
      NSLog(@"One argument expected.\n");
   }

   return 0;
}
```

This chapter provided a crash tutorial on environmental setup, basic syntax, data types, loops, functions, strings, and error handling.

BIBLIOGRAPHY

1. Objective-C Tutorial – https://www.tutorialspoint.com/objective_c/index.htm, accessed on May 4, 2022.
2. About Objective-C – https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html, accessed on May 4, 2022.

3. Difference between C and Objective-C – https://www.geeksforgeeks.org/difference-between-c-and-objective-c/, accessed on May 4, 2022.
4. Objective-C – Toolshttps://educationecosystem.com/guides/programming/objective-c/history, accessed on May 4, 2022.
5. Object-Oriented Programming in Objective-C – https://medium.com/ios-objective-creation/lesson-1-introduction-to-objective-c-programming-22f5fe71172, accessed on May 4, 2022.

CHAPTER 2

OOP in Objective-C

IN THIS CHAPTER

- Objects and Classes
- Class Patterns and Clusters
- Object Properties
- Additional OOP

In the previous chapter, we discussed the crash course of Objective-C and in this chapter, we will cover OOP's concept.

OBJECT-ORIENTED PROGRAMMING

As humans, we are constantly confronted with data and perceptions that we must interpret. We must separate the underlying structure from the surface features and identify the critical relationships at work. Abstractions explain causes and consequences, uncover patterns and frameworks, and distinguish between what is and is not relevant. Object orientation offers an abstraction of the data on which you work; moreover, it gives a tangible grouping between the data and the operations performed on the data – giving the data behavior.

OPERATIONS AND DATA

Data and actions on data have typically been separated in programming languages. Except when operations affect it, data is static and immutable.

DOI: 10.1201/9781003357506-2

Procedures and functions that act on data don't have a permanent state of their own; they're solely helpful for affecting data.

This separation is, of course, based on how computers operate, so it's not something you can lightly dismiss. It provides the backdrop against which we labor, just as the differences between matter and energy and nouns and verbs do. All programmers, even object-oriented programmers, must specify the data structures that their programs will utilize and the functions that will operate on the data at some point.

This is all there is to it in a procedural programming language like C. The language may provide multiple forms of data and function organization, but it will not divide the world. The fundamental parts of the design are functions and data structures.

Object-oriented programming restructures the world at a higher level rather than disputing it. It divides processes and data into objects, then combined into organized networks to produce a complete program. Objects and object interactions are the fundamental design components of an object-oriented programming language.

Every object contains state (data) and behavior (operations on data). They are similar to conventional physical things in this regard. A mechanical item, such as a pocket watch or a piano, may effortlessly embody state and behavior. But practically everything intended to do a task does. Even basic objects with no moving elements, such as a bottle, integrate state (the amount of liquid in the bottle, whether it is open, and how warm the contents are) with behavior (the ability to dispense its contents at various flow rates, to be opened or closed, to withstand high or low temperatures).

The force and attractiveness of items come from their similarity to actual things. They can not only represent real-world components, but they can also perform specified tasks as software components.

IMPLEMENTATION AND INTERFACE

We must be able to grasp abstractions and express them in program design to create programs. A programming language's role is to assist us in accomplishing this. The language should make it easier to innovate and design by allowing you to represent abstractions that reveal how things function. It should allow us to concretize our thoughts in the code we write. Surface details should not obscure your program's architecture.

All programming languages have tools for expressing abstractions. In essence, these devices are methods of collecting implementation

details, concealing them, and providing them with a common interface, in the same way, that a mechanical item separates its interface from its implementation.

Structures and functions are the primary abstraction units of the C language. Both conceal implementation components in various ways:

C structures aggregate data pieces into bigger units that may be treated as separate entities on the data side. While some programs must dive into the structure and alter the fields individually, most programs may treat it as a single entity, not a collection of components but the sum of those elements. Because one structure may include others, a complex information arrangement can be constructed from simpler levels.

In modern C, a structure's fields have their own namespace, so their names won't clash with similarly named data objects outside the structure. Keeping implementation details out of the interface requires partitioning the program namespace. Consider the tremendous challenge of giving each item of data in an extensive program a unique name while also ensuring that new names do not clash with existing ones.

Functions encapsulate activities that are used frequently without having to be re-implemented in the procedural world. Like data items local to a function, fields inside a structure are protected under their namespace. Complex actions are created from simpler elements because functions can reference (call) other functions.

Functions are reused. They are called any number of times after being defined without thinking about the implementation again. The most often used functions may be grouped into libraries and utilized across various applications. The function interface is all required of the user, not the source code.

Functions, unlike data components, are not divided into different namespaces. Every function must have its own name. While the function itself is reusable, the name is not.

Although C structures and functions may express large abstractions, they maintain the separation between data and data operations. The most outstanding levels of abstraction in a procedural programming language nevertheless reside on one side or the other of the data-versus-operations split. The way the computer operates is always reflected in the applications you create.

Object-oriented programming languages don't give up any benefits of structures and functions; instead, they add a higher-level abstraction unit that conceals the interaction between a function and its data.

Assume you have a collection of functions that interact with a specific data structure. You want to make those functions easy to use by removing the structure from the interface as much as feasible. So you add a couple more functions to help handle the data. The functions are responsible for allocating memory for the data structure, initializing it, retrieving information from it, changing values inside it, keeping it up to date, and clearing its memory. All the users of Objective-C have to do is call the functions and provide them with the structure.

The structure has become an opaque token due to these modifications, and other programmers will never need to see inside. They may focus on the functions rather than the data organization. You've started the process of making an item.

The next step is to implement this concept in a programming language and entirely conceal the data structure so that it is not required to be provided between functions. All that is exposed to consumers is a functional interface; the data becomes an internal implementation detail. Users may conceive of objects based on their behavior since they encapsulate (hide) their data.

The interface to the functions has been much simplified due to this stage. Callers aren't required to understand how they work (what data they use). This may now be classified as an item.

All functions that have access to the secret data structure are grouped. As a result, an object is more than a collection of random functions; it's a collection of connected behaviors backed by common data. To utilize an object's function, we must first build the object (give it its internal data structure) and then tell it the function it should execute. You start thinking about the object's overall function rather than the individual functions.

Learning object-oriented programming is all about moving from thinking about functions and data structures to thinking about object behaviors. It may seem strange at first, but as you acquire expertise with object-oriented programming, we'll discover that it's a more natural way of thinking. Lists, containers, tables, controllers, and even managers are all analogous to real-world objects in programming. Using programming objects as an example simply expands the comparison naturally.

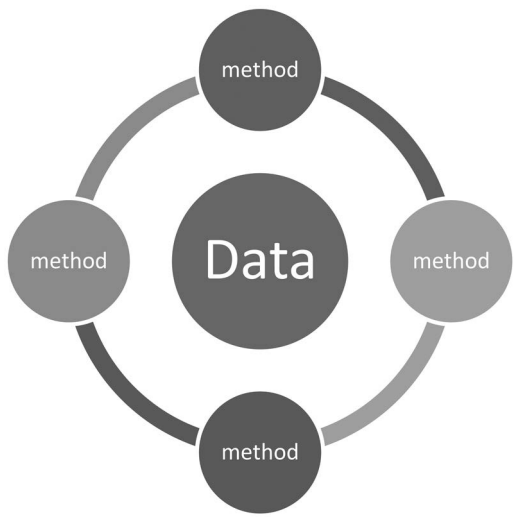
The types of abstractions that a programming language allows us to encode may be used to evaluate it. Extraneous issues should not distract us, nor should we be obliged to explain ourselves using a language that does not correspond to the reality we're attempting to depict.

If, for example, you must constantly attend to the business of matching the correct data with the correct method, you are compelled to be aware of the complete program at a low level of implementation at all times. While we may still create programs with a high degree of abstraction, the transition from concept to implementation can be challenging and increasingly tricky as programs become more complex.

Object-oriented programming languages provide us with a more extensive vocabulary and a richer model to work in by giving a greater degree of abstraction.

THE OBJECT MODEL

The object-oriented programming insight is to integrate state and behavior data and data operations into a high-level unit, an object, and provide language support. A collection of related functions and a data structure that fulfils those functions form an object. The functions in Objective-C are known as the object's methods, and the data structure's fields are known as its instance variables.



Model of object.

If we've ever worked on a challenging programming problem, our design certainly incorporated sets of functions that work on a specific type of data implicit "objects" without language support.

Object-oriented programming makes these function groups clear and allows us to think about the group rather than its components. The only

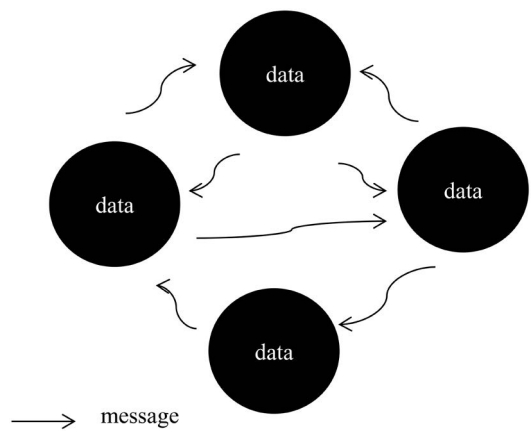
way to access an object's data, and hence the only interface, is via its methods.

When both state and behavior are combined in a single thing, it becomes more than either alone; the whole is greater than the sum of its parts. An object is a self-contained "subprogram" having authority over a specific functional area. It can function as a full-fledged modular component inside a bigger program design.

For example, if we were to create software that simulated residential water usage, we might create objects to represent the various components of the water-delivery system. One example is a Faucet object, which has methods for starting and stopping the water flow, adjusting the flow rate, returning the quantity of water consumed in a specific period, and so on. A Faucet object would require instance variables to track whether the tap is open or closed, how much water is being used, and where the water is coming from to perform this function.

A programmatic Faucet object can be wiser than a real one (it's like a mechanical faucet with many gauges and instruments). But, like any other system component, a genuine faucet has both state and behavior. We'll need programming units, like objects, which mix state and behavior, to represent a system successfully.

A program comprises a network of linked items that work together to solve a puzzle piece (as illustrated in the below image). Each object has a defined function in the program's general architecture and may connect with other objects. Objects interact through messages, which are requests for methods to be performed.



Network in objects.

The network's objects will not be identical. In addition to Faucet objects, a program that replicates water use may include Pipe objects that carry water to the Faucet and Valve objects that control the flow between pipes. There may be a Building object to coordinate a system of pipes, valves, and faucets, some Appliance objects that can switch valves on and off (equivalent to dishwashers, toilets, and washing machines), and perhaps some User objects to operate the appliances and faucets. When a Building object in Objective-C is asked how much water is being utilized, it may request information from each Faucet and Valve object. When a user turns on an appliance, it must first turn on a valve to get the water it needs.

THE METAPHOR OF MESSAGING

Each programming paradigm has its own set of terms and metaphors. The vocabulary of object-oriented programming encourages you to consider what happens in a program from a unique viewpoint.

There's a temptation, for example, to consider things to be actors with human-like motivations and capacities. It's tempting to speak about an object determining what to do in a scenario, requesting information from other objects, introspecting to get the required information, transferring responsibilities to another object, or managing a process.

This metaphor requires you to conceive objects as executing their methods rather than functions or methods doing the job, as you would in a procedural programming language. Objects are the agents of the program's action rather than passive receptacles for state and behavior.

This metaphor is quite helpful. An object is similar to an actor in specific ways: it has a specific job to perform in the program's overall design. Within that role, it may work somewhat independently of the other components. It interacts with other things as they perform their functions, yet it is self-contained and may operate independently. It can't deviate from the script, just like a performer onstage, yet the part it performs might be diverse and intricate.

Objects as actors mesh well with the primary metaphor of object-oriented programming, which is that objects interact through messages. Instead of executing a method like a function, we send a message to an object and ask it to do one of its methods.

This metaphor leads to a beneficial way of thinking about methods and objects; however, it takes some getting accustomed to it. It isolates procedures from the specific data they operate on and instead focuses on

behavior. A start method, for example, in an object-oriented programming interface, may start an operation, an archive method, and a draw method, for example, might output an image. The method name does not tell which action is started, which information is saved, or which picture is drawn. These procedures may be performed differently by distinct objects.

Methods are, therefore, a vocabulary of abstract activities. To make one of those behaviors concrete, you must associate the method with an object. This is accomplished by designating the object as the message's recipient. The process that is started, the data archived, and the rendered picture are all determined by the item you choose as the recipient.

Methods are called via a specific receiver since they are objects (the owner of the method and the data structure the method will act on). The same procedure might be implemented differently by different receivers. As a result, various receivers might respond to the same message differently. A message's outcome can't be determined just based on the message or method name; it additionally relies on the object that receives the message.

The messaging metaphor nicely captures the concept that actions may be abstracted from their specific implementations by separating the message (the requested behavior) from the receiver (the owner of a method that can react to the request).

Objective-C offers considerable support for designing object-oriented iOS iPad apps. However, the field of object-oriented programming is vast. It is not hyperbole to say that whole volumes have been written on the subject. As a result, a comprehensive review of object-oriented software development is outside the scope of this book. Instead, we will present the fundamental notions of object-oriented programming before describing the concept as it applies to Objective-C application development. Again, while we aim to offer the essential knowledge you need in this chapter, if we are inexperienced with Objective-C programming, we recommend reading a copy of Objective-C 2.0 Essentials.

The main goal of the Objective-C programming language is to add object orientation to the C programming language, and classes, also known as user-defined types, are the central feature of Objective-C that support object-oriented programming.

A class is used to specify an object's form, and it combines data representation and data manipulation methods into a single package. Members of a class are the data and methods contained within it.

CHARACTERISTIC OF OBJECTIVE-C

- @interface and @implementation are the two sections that define the class.
- Almost everything takes the shape of an object.
- Objects are often referred to as receivers because they receive messages.
- Instance variables are found in objects.
- Scope exists in objects and instance variables.
- Classes hide the implementation of an object.
- Properties are used to give other classes access to class instance variables.

DEFINITIONS OF OBJECTIVE-C CLASSES

We create a blueprint for a data type when you create a class. This doesn't define any data, but it does define what the class name means, that is, what a class object will be made up of and what operations are performed on it.

The @interface keyword is followed by the interface(class) name and the class body, enclosed by a pair of curly braces. All classes in Objective-C are derived from the base class NSObject. All Objective-C classes are derived from it. It includes basic memory allocation and initialization methods. For example, using the keyword class, we defined the Box data type as follows:

```
@interface Box:NSObject {
   //Instance-variables
   double length;    // box Length
   double breadth;   // box Breadth
}
@property(nonatomic, readwrite) double height;  //
Property

@end
```

ALLOCATING AND INITIALIZING OBJECTIVE-C OBJECTS

A class supplies the blueprints for objects; hence an object is formed from a class. We define objects of a class in the same way as we declare variables of fundamental kinds. The statements that follow declare two Box objects.

```
Box box1 = [[Box alloc]init];      // Create the box1
object of the type Box
Box box2 = [[Box alloc]init];      // Create the box2
object of the type Box
```

ACCESSING DATA MEMBERS

The direct member access operator is used to access the properties of class objects (.). To illustrate, consider the following example:

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

@interface Box:NSObject {
   double length;    // box Length
   double breadth;   // box Breadth
   double height;    // box Height
}

@property(nonatomic, readwrite) double height;  //
Property
-(double) volume;
@end

@implementation-Box

@synthesize height;

-(id)init {
   self = [super init];
   length = 2.0;
   breadth = 2.0;
   return self;
}

-(double) volume {
   return length*breadth*height;
}
```

```
@end

int main() {
   NSAutoreleasePool * pool = [[NSAutoreleasePool
alloc] init];
   Box *box1 = [[Box alloc]init];    // Create the
box1 object of the type Box
   Box *box2 = [[Box alloc]init];    // Create the
box2 object of the type Box

   double volume = 0.0;             // Store volume of
box here

   // box1 specification
   box1.height = 6.0;

   // box2 specification
   box2.height = 12.0;

   // volume of the box 1
   volume = [box1 volume];
   NSLog(@"Volume of the Box1 : %f", volume);

   // volume of the box 2
   volume = [box2 volume];
   NSLog(@"Volume of the Box2 : %f", volume);

   [pool drain];
   return 0;
}
```

Properties

- In Objective-C, properties ensure that a class's instance variable is accessed from outside the class.
- The following are the various parts of the property declaration.
- @property is a keyword that starts properties.
- It's followed by access specifiers such as nonatomic, atomic, read-write, and readonly, strong, unsafe unretained, and weak. This differs depending on the variable's type. We can use unsafe unretained,

strong, or weak for any pointer type. We may also use readwrite or readonly for other types.

- The variable's datatype follows this.
- A semicolon finally concludes the property name.
- In the implementation class, we can add a synthesize statement. However, in the most recent version of XCode, the program handles the synthesis, and the synthesize statement is no longer required.

The only way to access the class's instance variables is through the properties. For the properties, getter and setter methods are built internally.

For example, consider the property @property (nonatomic, readonly) BOOL isDone. Setters and getters have been constructed under the hood, as illustrated below.

```
-(void)setIsDone(BOOL)isDone;
-(BOOL)isDone;
```

Modularity

A module in Objective-C is nothing more than a file containing source code to a C programmer. A practical technique to divide a vast (or even not-so-large) program into manageable bits is to break it down into separate files. Each item may be worked on and compiled separately and then connected after the program is completed.

Using the static storage class designator to restrict the scope of names to the files where they are declared improves source module independence. This kind of module is a file system-defined unit. It's a container for source code rather than a logical language unit. Each coder decides what gets inside the container. We don't have to use them to group logically similar pieces of the code. We may put your socks in one drawer, underwear in another, and so on, or you can use another organization system or jumble everything up.

Object-oriented programming languages allow us to utilize file containers for your source code, but they also have a logical module called class definitions. As you would anticipate, each class is often specified in its source file. Logical modules are paired with container modules.

In Objective-C, for example, the component of the Valve class that interacts with Pipe objects may be defined in the same file as the Pipe class, resulting in a container module for Pipe-related code and the Valve class

being divided into several files. No matter how many files the source code was in, the Valve class definition would still work as a modular unit inside the program's construction. It would still be a logical module.

Under "Processes of Abstraction," the mechanisms that make class definitions logical components of the language are addressed in depth.

Reusability

In Objective-C one of the main goals of object-oriented programming is to make the code you create as reusable as possible – to have it serve many various scenarios and applications – so that you don't have to re-implement something that has previously been done even if just slightly differently.

The following elements impact reusability: how stable and bug-free the code is; how clear the documentation is; how simple and easy the programming interface is; how efficiently the code accomplishes its job; and how complete the feature set is.

These considerations aren't limited to the object model. They are used to determine the reusability of any code, including standard C functions and class declarations. Functions that are efficient and well-documented, for example, are more reusable than those that are undocumented and unreliable.

Nonetheless, a broad comparison reveals that class definitions favor reusable code in ways that functions do not. There are many ways to make functions more reusable, such as giving data as arguments rather than assuming particular global variables. Despite this, only a tiny portion of functions is extended beyond the purposes they were developed. In at least three ways, their reusability is constrained by design:

- The names of functions are global; each function must have its own name (except for those declared static). This naming restriction makes it difficult to depend extensively on library code when developing a complicated system. The programming interface would be difficult to grasp and so large that significant generalizations would be challenging to express.

  In contrast, classes may share programming interfaces. When the same naming conventions are used repeatedly, a large amount of functionality is packed into a compact, easy-to-understand interface.

- One by one, functions are chosen from a library. It is up to programmers to choose the specific functions they need.

- On the other hand, objects are functional bundles rather than individual methods and instance variables. Users of an object-oriented library in Objective-C won't get bogged down putting together their answers to a problem since they offer integrated services.
- Functions are usually linked to unique data structures created for a given application. Data and function interaction is an inevitable aspect of the interface. Only those who agree to utilize the same data structures as the function's arguments will find it beneficial.

  An object does not have this difficulty since it conceals its data. One of the main reasons classes is reused more readily than functions is this.

An object's data is secured and will not be accessed by any other portion of the program. As a result, methods may rely on their reliability. They can be confident that data has not become irrational or unusable due to external access. Consequently, a data structure supplied to an object is more dependable than one passed to a function, and methods may rely on it more. As a result, reusable methods are easy to develop.

A class may also be re-implemented to utilize a new data structure without altering its interface since an object's data is concealed. All applications that utilize the class may update to the new version without reprogramming; no source code changes are necessary.

INHERITANCE IN OBJECTIVE-C

Starting with something familiar is the most straightforward approach to communicating something unfamiliar. If we're describing a schooner, it helps if our audiences are familiar with sailboats. Suppose we're explaining how a harpsichord works. In that case, it's helpful if we can presume our audience has seen a piano inside or heard a guitar played, or at the very least is acquainted with the concept of a musical instrument.

The same is true if we want to describe a new kind of object; starting from the definition of an existing object simplifies the description.

Given this, object-oriented programming languages allow us to base a new class definition on an existing one. A superclass is the base class, while a subclass is a new class. Only how the subclass varies from the superclass is specified in the subclass specification; everything else is assumed to be the same.

In object-oriented programming, inheritance is one of the most important concepts. Inheritance in Objective-C allows us to define a class in terms of another class, making application development and maintenance easier. This also allows for the reuse of code functionality and a quick implementation time.

Instead of developing entirely new data members and member functions when creating a class, the programmer can specify that the new class should inherit the members of an existing class. The old class is the base class, while the new class is the derived class.

The concept of inheritance establishes a connection – for instance, mammal IS-A animal, dog IS-A mammal, dog IS-A animal, etc.

BASE AND DERIVED CLASSES

Objective-C only supports multilevel inheritance, which means that it may have just one base class but supports multilevel inheritance. All Objective-C classes are derived from the superclass NSObject.

```
@interface derivedclass: baseclass
```

Consider the following for a base class Person and its derived class Employee:

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

@interface Person : NSObject {
   NSString *personName;
   NSInteger personAge;
}

- (id)initWithName:(NSString *)name andAge:(NSInteger)
age;
- (void)print;

@end

@implementation Person

- (id)initWithName:(NSString *)name andAge:(NSInteger)
age {
   personName = name;
   personAge = age;
```

```
   return self;
}

- (void)print {
   NSLog(@"The Name is: %@", personName);
   NSLog(@"The Age is: %ld", personAge);
}

@end

@interface Employee : Person {
   NSString *employeeEducation;
}

- (id)initWithName:(NSString *)name andAge:(NSInteger)
age
  andEducation:(NSString *)education;
- (void)print;
@end

@implementation Employee

- (id)initWithName:(NSString *)name andAge:(NSInteger)
age
   andEducation: (NSString *)education {
      personName = name;
      personAge = age;
      employeeEducation = education;
      return self;
   }

- (void)print {
   NSLog(@"The Name is: %@", personName);
   NSLog(@"The Age is: %ld", personAge);
   NSLog(@"Education: %@", employeeEducation);
}

@end

int main(int argc, const char * argv[]) {
   NSAutoreleasePool * pool = [[NSAutoreleasePool
alloc] init];
   NSLog(@"The Base class Person Object");
   Person *person = [[Person alloc]
initWithName:@"Rajat" andAge:15];
```

```
    [person print];
    NSLog(@"Inherited Class Employee Object");
    Employee *employee = [[Employee alloc]
initWithName:@"Rajat"
    andAge:15 andEducation:@"BA"];
    [employee print];
    [pool drain];
    return 0;
}
```

ACCESS THE CONTROL AND INHERITANCE

If a derived class is defined in the interface class, it can access all of its base class's private members, but it cannot access those defined in the implementation file.

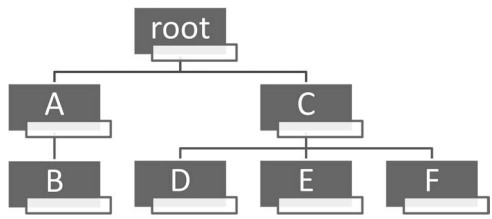
The different access kinds are summarized in the following fashion based on who has access to them.

With the exceptions indicated below, a derived class inherits all base class methods and variables.

- Variables declared with the help of extensions in the implementation file are not accessible.
- Methods declared with the help of extensions in the implementation file are not accessible.
- The derived class method is run if the inherited class implements the base class method.

Hierarchies of Class

Any class in Objective-C can be used as a superclass for defining a new class. A class can be a subclass of another class and a superclass for its subclasses. As a result, any number of classes can be linked in a hierarchy of inheritance, as shown in Figure.



Hierarchy of inheritance.

Starting with something familiar is the most straightforward approach to communicating something unfamiliar. If we're describing a schooner, it helps if our audience is familiar with sailboats. Suppose we're explaining how a harpsichord works. In that case, it's helpful if we can assume our audience has seen a piano inside or heard a guitar played, or at the very least is acquainted with the concept of a musical instrument.

The same is true if we want to describe a new kind of object; starting from the definition of an existing object simplifies the description.

Given this, object-oriented programming languages allow us to base a new class definition on an existing one. A superclass is the base class, while a subclass is a new class. Only how the subclass varies from the superclass is specified in the subclass specification; everything else is assumed to be the same.

Definitions of Subclass

A subclass may update the definition it inherits from its superclass in three ways.

It may add additional methods and instance variables to the class definition it inherits. The most typical motivation for creating a subclass is to solve a problem. Subclasses always add additional methods and instance variables if the methods demand it.

Replacing an existing method in Objective-C with a new one might change the behavior it inherits. This is accomplished by simply creating a new method with the same name as an inherited one. The new version takes precedence over the previous one. (The inherited method remains valid for the class that defined it and any classes that inherit it.)

It may improve or expand the behavior it inherits by replacing an existing method with a new one, but it keeps the old one by integrating it into the new one. In the body of the new method, a subclass sends a message to execute the old version. Each class in an inheritance chain may influence the behavior of a method in some way. Class D, for example, may override a method specified in class C and integrate C's version, while C's version incorporates a version declared in the root class, as shown in Figure.

Subclasses tend to round out the definition of a superclass, making it more particular and specialized. Instead of subtracting code, they add and occasionally change it. It's worth noting that methods and instance variables can't be deleted or overridden in most cases.

DYNAMISM

The issue of how much memory a program will take was once resolved when the source code was built and linked. As soon as the program begins, it is given the entire RAM it would ever need. This memory could not be expanded or contracted.

In hindsight, it's clear that this was a significant constraint. It restricted how programs are built and what a program could perform. It limited not just the programming approach but also the design. The development of methods like malloc, which dynamically allocates memory while a program runs, opened up previously unimaginable possibilities.

Compile-time and link-time limitations are restrictive because they require decisions based on information discovered in the programmer's source code rather than information gained from the user as the program runs.

Although dynamic memory allocation eliminates one such limitation, many others remain, just as restrictive as static memory allocation. For example, at build time, the parts that make up an application must be matched to data types. And an application's bounds are usually specified at link time. The whole program contains inside a single executable file. New modules and types are not addable while the application is running.

The goal of Objective-C is to get beyond these limits and make programs as dynamic and fluid as possible. It moves most decision-making efforts from build and link time to runtime. The idea lets programmers determine what happens rather than artificially limiting their activities by language demands and compiler and linker requirements.

For object-oriented design, three types of dynamism are essential:

- Dynamic typing involves waiting until runtime to determine an object's class.
- Dynamic binding involves determining which method to invoke at runtime.
- Dynamic loading involves adding new components to a program as it runs.

POLYMORPHISM IN OBJECTIVE-C

Polymorphism is the ability of different things to react to identical signals uniquely.

The fact that each class has its own namespace leads to polymorphism. The names assigned inside a class definition are not incompatible with names given elsewhere. This is true for both instance variables in an object's data structure and its methods.

An object's instance variables, like the fields of a C structure, are in a protected namespace, as are method names. Method names, unlike C function names, are not global symbols. A method name in one class cannot clash with the names of methods in other classes; two classes with completely distinct names may implement identically named methods.

The interface of an object includes method names. When a message instructs an object to complete a task, the message specifies the method the object should use. Because several objects might have methods with the same name, a message's meaning is understood concerning the object that receives it. The same message might trigger two different methods when delivered to two different objects.

Polymorphism's key advantage is that it simplifies the programming interface. It enables the establishment of conventions that are reused from class to class. Rather than establishing a new name for each new function, you add to a program. We may reuse existing names. Separate from the classes that implement them, the programming interface may be defined as a collection of abstract behaviors.

Polymorphism is defined as having several forms. Polymorphism often happens when a hierarchy of classes is connected via inheritance.

Because of Objective-C polymorphism, a call to a member function will run a different code depending on the kind of object that invokes the function.

Consider the following scenario: we have a class Shape that offers the fundamental interface for all forms. The shape is the foundation class from which squares and rectangles are formed.

We have the function printArea, which will display information about the OOP feature polymorphism.

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

@interface Shape : NSObject {
   CGFloat area;
}

<CODE>- (void)printArea;
```

```
- (void)calculateArea;
@end

@implementation Shape
- (void)printArea {
   NSLog(@"Area is %f", area);
}

- (void)calculateArea {

}

@end

@interface Square : Shape {
   CGFloat length;
}

- (id)initWithSide:(CGFloat)side;
- (void)calculateArea;

@end

@implementation Square
- (id)initWithSide:(CGFloat)side {
   length = side;
   return self;
}

- (void)calculateArea {
   area = length * length;
}

- (void)printArea {
   NSLog(@"The Area of square is %f", area);
}

@end

@interface Rectangle : Shape {
   CGFloat length;
   CGFloat breadth;
}
```

```
- (id)initWithLength:(CGFloat)rLength
andBreadth:(CGFloat)rBreadth;
@end

@implementation Rectangle
- (id)initWithLength:(CGFloat)rLength
andBreadth:(CGFloat)rBreadth {
   length = rLength;
   breadth = rBreadth;
   return self;
}

- (void)calculateArea {
   area = length * breadth;
}

@end

int main(int argc, const char * argv[]) {
   NSAutoreleasePool * pool = [[NSAutoreleasePool
alloc] init];
   Shape *square = [[Square alloc]initWithSide:20.0];
   [square calculateArea];
   [square printArea];
   Shape *rect = [[Rectangle alloc]
   initWithLength:20.0 andBreadth:15.0];
   [rect calculateArea];
   [rect printArea];
   [pool drain];
   return 0;
}
```

In the above example, depending on the availability of the methods calculateArea and printArea, either the method from the base class or the method from the derived class was performed.

Polymorphism manages method switching between the base and derived classes based on their method implementations.

DATA ENCAPSULATION IN OBJECTIVE-C

We must be able to leave implementation details behind and conceptualize in terms of units that organize those elements under a standard interface to design successfully at any degree of abstraction. The barrier between

interface and implementation must be absolute for a programming unit to be genuinely effective. The implementation must be encapsulated in the interface, hiding it from other portions of the program. Encapsulation shields an implementation against unintended consequences and unauthorized access.

A function in C is explicitly encapsulated; its implementation is unavailable to other portions of the program and shielded from activities outside the function's body. Method implementations are similarly enclosed in Objective-C, but an object's instance variables are more significant. They're concealed inside the thing yet visible from the outside. Information hiding is another term for the encapsulation of instance variables.

At first glance, concealing information in instance variables may limit our programming flexibility. It provides you with greater leeway to act and frees us from restrictions that we may otherwise face. If any component of an object's implementation leaks out and becomes accessible or a concern to other program sections, it ties the implementer's and users' hands. Neither could make changes without consulting the other first.

We're interested in the Faucet object built for a program that mimics water usage and wants to utilize it in another program we're building. Once we've agreed on the object's interface, we won't have to worry about others working on it, fixing issues, and finding better implementation methods. These enhancements help us, but none of them change what we do in our program. Nothing they do can damage our code since we rely entirely on the interface. The implementation of the object is isolated from our software.

Furthermore, although individuals developing the Faucet object may be curious about how we use it and want to make sure it fits our requirements, they aren't concerned with how we write our code. Nothing we do will affect the object's implementation or restrict its ability to make modifications in future versions. The implementation is protected from whatever other object users or we could do.

The following two key features are included in all Objective-C programs.

Methods are the parts of a program that perform actions, and they have termed program statements (code).

Program data is the information about the program that is influenced by its functions.

Encapsulation is an Object-Oriented Programming concept that connects data and the functions that handle it, keeping them safe from outside influence and misuse. The crucial OOP notion of data hiding was initially developed from data encapsulation.

Data encapsulation and data abstraction are two mechanisms for bundling data and its functions. Data encapsulation exposes only the interfaces while hiding the implementation details from the user.

By constructing user-defined kinds or classes, Objective-C enables the properties of encapsulation and data hiding. For instance,

```
@interface Adder : NSObject {
   NSInteger total;
}

- (id)initWithInitialNumber:(NSInteger)initialNumber;
- (void)addNumber:(NSInteger)newNumber;
- (NSInteger)getTotal;

@end
```

Total is a private variable that we can't access from outside the class. This means that they can only be accessible by other Adder class members and not by the other part of our program. This is one method of encapsulation.

Methods in the interface file are public in scope and accessible.

EXAMPLE OF DATA ENCAPSULATION

Data encapsulation and data abstraction may be seen in any Objective-C program that implements a class with public and private member variables. Consider the following situation:

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

@interface Adder : NSObject {
   NSInteger total;
}

- (id)initWithInitialNumber:(NSInteger)initialNumber;
- (void)addNumber:(NSInteger)newNumber;
- (NSInteger)getTotal;

@end

@implementation Adder
-(id)initWithInitialNumber:(NSInteger)initialNumber {
   total = initialNumber;
```

```
    return self;
}

- (void)addNumber:(NSInteger)newNumber {
    total = total + newNumber;
}

- (NSInteger)getTotal {
    return total;
}

@end

int main(int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool
alloc] init];
    Adder *adder = [[Adder alloc]
initWithInitialNumber:10];
    [adder addNumber:5];
    [adder addNumber:4];

    NSLog(@"Total is %ld",[adder getTotal]);
    [pool drain];
    return 0;
}
```

The total of the numbers in the above class is returned. The public members addNum and getTotal are class's external interfaces, and a user must be familiar with them to utilize them. The private member total is hidden from the outside world, but it is required for the class to function correctly.

CREATING A STRATEGY

Most of us have come to the realization that unless we have a compelling reason, we should make class members private by default. That is an amazing encapsulation.

Data encapsulation is crucial because it is one of the critical aspects of all Object-Oriented Programming (OOP) languages, including Objective-C.

CATEGORIES IN OBJECTIVE-C

We might want to extend an existing class by adding behavior that is only useful in specific circumstances. Objective-C provides categories and extensions for adding such extensions to existing classes.

The simplest way to add a method to an existing class, perhaps to add functionality to make something easier to do in your application, is to use a category.

The @interface keyword in Objective-C is used to declare a category, just as it is in a standard Objective-C class description, but it does not indicate any inheritance from a subclass. Instead, it uses parentheses to specify the category name, such as this:

```
@interface Class_Name (Category_Name)

@end
```

CATEGORY CHARACTERISTICS

Even if we don't have the original implementation source code, we can establish a category for any class in Objective-C.

Any methods declared in a category in Objective-C will be available to all instances of the original class and any subclasses.

There is no difference between a category-added method and one implemented by the original class at runtime.

Let's have a look at an example of a category implementation. Let's give the Cocoa class NSString a category. This category will allow us to add a new method called getCopyRightString to return the copyright string. It is displayed below.

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

@interface NSString(MyAdditions)
+(NSString *)getCopyRightString;
@end

@implementation NSString(MyAdditions)

+(NSString *)getCopyRightString {
   return @"Copyright Point.com 2022";
}

@end

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

```
    NSString *copyrightString = [NSString
getCopyRightString];
    NSLog(@"Accessing Category: %@",copyrightString);

    [pool drain];
    return 0;
}
```

Even though any methods introduced by a category are available to all class instances and subclasses, we must import the category header file in any source code file where we want to utilize the new methods. Otherwise, you may encounter compiler warnings and failures.

Because we have a single class in our example, we haven't included any header files; in this situation, we need to include the header files mentioned above.

POSING IN OBJECTIVE-C

Before discussing Posing in Objective-C, it's important to note that posing was declared deprecated in Mac OS X 10.5 and is no longer accessible for use. Those who are unconcerned about the deprecated methods can therefore skip this chapter.

A class in Objective-C can completely replace another class in a program. It is argued that the replacing class "poses as" the target class. Instead, the posing class receives all messages sent to the target class in versions that allow posing.

As shown above, the poseAsClass method in NSObject allows us to replace the existing class.

POSING RESTRICTIONS

- Only one of a class's direct or indirect superclasses is used.
- No new instance variables are defined in the posing class not present in the target class (though it may define or override methods).
- Before the posing, the target class may not have received any messages.
- A posing class can use super to call overridden methods, integrating the target class's implementation.
- A posing class in Objective-C can override methods defined in categories.

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

@interface MyString : NSString

@end

@implementation MyString

- (NSString *)stringByReplacingOccurrencesOfString:(NS
String *)target
withString:(NSString *)replacement {
   NSLog(@"Target string: %@",target);
   NSLog(@"Replacement string: %@",replacement);
}

@end

int main() {
   NSAutoreleasePool * pool = [[NSAutoreleasePool
alloc] init];
   [MyString poseAsClass:[NSString class]];
   NSString *string = @"Test";
   [string stringByReplacingOccurrencesOfString:@"a"
withString:@"c"];

   [pool drain];
   return 0;
}
```

EXTENSIONS IN OBJECTIVE-C

A class extension in Objective-C is similar to a category, except it can only be applied to a class for which the source code is available at compile time (the class is compiled simultaneously).

We can't declare a class extension on a framework class, such as a Cocoa or Cocoa Touch class like NSString since the methods declared by a class extension are implemented in the implementation block for the original class.

Extensions are simply categories that don't have a name. It's also known as "anonymous categories."

@interface keyword is used to declare an extension, precisely like it is in a regular Objective-C class description, but it does not show any inheritance from a subclass. Instead, as illustrated below, it just adds parenthesis.

```
@interface ClassName ()

@end
```

EXTENSIONS' CHARACTERISTICS

- An extension is only declared for classes for which we know the original source code implementation.
- A class extension adds private methods and variables only available to that class.
- Even the inherited classes cannot access any methods or variables declared inside the extensions.

Example of Extensions

Let's make a class called Sample_Class with an extension. Let's have a private variable internalID in the extension.

Then, after processing the internalID, create a method called getExternalID that returns the externalID.

The following example will not work with an online compiler.

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

@interface Sample_Class : NSObject {
   NSString *name;
}

- (void)setInternalID;
- (NSString *)getExternalID;

@end

@interface Sample_Class() {
   NSString *internalID;
}

@end
```

```
@implementation Sample_Class

- (void)setInternalID {
   internalID = [NSString stringWithFormat:
   @"UNIQUEINTERNALKEY%dUNIQUEINTERNALKEY",arc4ran
dom()%100];
}

- (NSString *)getExternalID {
   return [internalID
stringByReplacingOccurrencesOfString:
   @"UNIQUEINTERNALKEY" withString:@""];
}

@end

int main(int argc, const char * argv[]) {
   NSAutoreleasePool * pool = [[NSAutoreleasePool
alloc] init];
   Sample_Class *sample_Class = [[Sample_Class alloc]
init];
   [sample_Class setInternalID];
   NSLog(@"ExternalID: %@",[sample_Class
getExternalID]);
   [pool drain];
   return 0;
}
```

The internalID is not returned explicitly in the preceding case. We delete the UNIQUEINTERNALKEY and simply make the remaining value available to the getExternalID function.

The above example only employs a string operation, but it may include numerous features such as encryption and decryption.

PROTOCOLS IN OBJECTIVE-C

Objective-C allows us to design protocols that specify the techniques utilized in a specific context. Protocols are implemented in protocol-conforming classes.

A network URL handling class, for example, will have a protocol containing methods such as the processCompleted delegate method, which informs the caller class once the network URL fetching operation is complete.

The protocol syntax is provided below.

```
@protocol ProtocolName
@required
// list of the required methods
@optional
// list of the optional methods
@end
```

The methods with the keyword @required are implemented in the classes that adhere to the protocol, whereas the methods with the keyword @optional are optional.

Here is the syntax:

```
@interface MyClass : NSObject <MyProtocol>
.....
@end
```

This implies that any instance of MyClass will react not just to the methods stated directly in the interface but also to the methods needed in MyProtocol. There is no need to redeclare the protocol methods in the class interface; simply implementing the protocol suffices.

If you want a class to use several protocols, describe them as a comma-separated list. We have a delegate object that retains the reference to the protocol-implementing calling object.

An example is shown below.

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

@protocol PrintProtocolDelegate
- (void)processCompleted;

@end

@interface PrintClass :NSObject {
   id delegate;
}

- (void) printDetails;
```

```
- (void) setDelegate:(id)newDelegate;
@end

@implementation PrintClass
- (void)printDetails {
   NSLog(@"Printing Details");
   [delegate processCompleted];
}

- (void) setDelegate:(id)newDelegate {
   delegate = newDelegate;
}

@end

@interface Sample_Class:NSObject<PrintProtocolDeleg
ate>
- (void)startAction;

@end

@implementation Sample_Class
- (void)startAction {
   PrintClass *printClass = [[PrintClass alloc]init];
   [printClass setDelegate:self];
   [printClass printDetails];
}

-(void)processCompleted {
   NSLog(@"The Printing Process Completed");
}

@end

int main(int argc, const char * argv[]) {
   NSAutoreleasePool * pool = [[NSAutoreleasePool
alloc] init];
   Sample_Class *sampleClass = [[Sample_Class alloc]
init];
   [sample_Class startAction];
   [pool drain];
   return 0;
}
```

We saw how the delegate methods are called and performed in the preceding example. It begins with startAction, and after the process is finished, the delegate method processCompleted is called to notify the action is finished.

We shall never have a program developed without a delegate in any iOS or Mac app. As a result, we must understand how delegates are used. To minimize memory leaks, delegate objects should utilize the unsafe_unretained property type.

DYNAMIC BINDING IN OBJECTIVE-C

Dynamic binding decides which method to call at runtime rather than compile time. Late binding is another term for dynamic binding. All methods in Objective-C are dynamically resolved at runtime. Both methods namely *the selector* and the receiving object decide the specific code run.

Polymorphism is enabled through dynamic binding. Consider the following set of objects: Rectangle and Square. A printArea method is implemented differently for each object.

The exact code that should be performed by the expression [anObject printArea] is decided at runtime in the following code snippet. The runtime system uses selector for the method run to find the proper method in whatever class anObject turns out to be.

Let's look at some basic code that demonstrates dynamic binding.

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

@interface Square:NSObject {
   float area;
}

- (void)calculateAreaOfSide:(CGFloat)side;
- (void)printArea;
@end

@implementation Square
- (void)calculateAreaOfSide:(CGFloat)side {
   area = side * side;
}

- (void)printArea {
   NSLog(@"The Area of square is %f",area);
}
```

```
@end

@interface Rectangle:NSObject {
   float area;
}

- (void)calculateAreaOfLength:(CGFloat)length
andBreadth:(CGFloat)breadth;
- (void)printArea;
@end

@implementation  Rectangle

- (void)calculateAreaOfLength:(CGFloat)length
andBreadth:(CGFloat)breadth {
   area = length * breadth;
}

- (void)printArea {
   NSLog(@"The Area of Rectangle is %f",area);
}

@end

int main() {
   Square *square = [[Square alloc]init];
   [square calculateAreaOfSide:20.0];

   Rectangle *rectangle = [[Rectangle alloc]init];
   [rectangle calculateAreaOfLength:20.0
andBreadth:15.0];

   NSArray *shapes = [[NSArray alloc]initWithObjects:
square, rectangle,nil];
   id obj1 = [shapes objectAtIndex:0];
   [obj1 printArea];

   id obj2 = [shapes objectAtIndex:1];
   [obj2 printArea];

   return 0;
}
```

As shown in the above example, the printArea method is dynamically determined during runtime. It is an example of dynamic binding and can be beneficial in various circumstances involving similar types of objects.

COMPOSITE OBJECTS IN OBJECTIVE-C

We may construct a subclass within a class cluster that creates a class that contains an object. These composite items are class objects. So we're probably wondering what a class cluster involves. So, initially, let's define a class cluster.

CLASS CLUSTERS

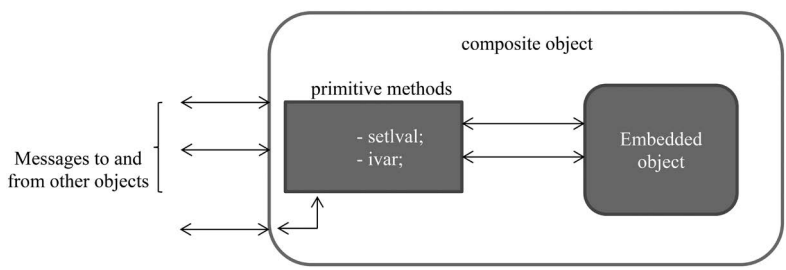
The foundation framework makes considerable use of class clusters as a design pattern. A collection of private concrete subclasses is grouped under a class cluster's public abstract superclass. This collection of classes simplifies an object-oriented framework's publicly visible design without decreasing its functional richness. The abstract factory design pattern in Objective-C is used to create class clusters.

Instead of developing many classes for comparable tasks, we construct a single class that will handle everything depending on the input value.

In NSNumber, for example, there are several clusters of classes such as char, int, bool, and so on. We combine them all into a single class that handles all of the related actions in one place. The value of these basic types is wrapped into objects by NSNumber.

WHAT EXACTLY IS A COMPOSITE OBJECT?

We build a composite object by embedding a private cluster object in an object of our design. This composite object in Objective-C can rely on the cluster object for basic functionality, intercepting messages that it wishes to handle in a certain way. This approach decreases the amount of code we must build and allows us to use the Foundation Framework's proven functionality.



Composite object.

The composite object in Objective-C must declare itself to be a subclass of the cluster's abstract superclass. It must override the primitive methods of the superclass as a subclass. It can also override derived methods, although this isn't required because the derived methods make their way via the primitive ones in Objective-C.

The count method of NSArray class is an example; the implementation of a method overridden by the intervening object might be as basic as

```
- (unsigned)count  {
   return [embedded_Object count];
}
```

An Example of a Composite Object

Kindly see the example from the Apple documentation provided below to see a comprehensive example.

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

@interface ValidatingArray : NSMutableArray {
   NSMutableArray *embeddedArray;
}

+ validatingArray;
- init;
- (unsigned)count;
- objectAtIndex:(unsigned)index;
- (void)addObject:object;
- (void)replaceObjectAtIndex:(unsigned)index
withObject:object;
- (void)removeLastObject;
- (void)insertObject:object atIndex:(unsigned)index;
- (void)removeObjectAtIndex:(unsigned)index;

@end

@implementation ValidatingArray
- init {
   self = [super init];
   if (self) {
      embeddedArray = [[NSMutableArray
allocWithZone:[self zone]] init];
   }
```

```
    return self;
}

+ validatingArray {
    return [[self alloc] init] ;
}

- (unsigned)count {
    return [embeddedArray count];
}

- objectAtIndex:(unsigned)index {
    return [embeddedArray objectAtIndex:index];
}

- (void)addObject:(id)object {
    if (object != nil) {
        [embeddedArray addObject:object];
    }
}

- (void)replaceObjectAtIndex:(unsigned)index
withObject:(id)object; {
    if (index <[embeddedArray count] && object != nil)
{
        [embeddedArray replaceObjectAtIndex:index
withObject:object];
    }
}

- (void)removeLastObject; {
    if ([embeddedArray count] > 0) {
        [embeddedArray removeLastObject];
    }
}

- (void)insertObject:(id)object atIndex:(unsigned)
index; {
    if (object != nil) {
        [embeddedArray insertObject:object
atIndex:index];
    }
}
```

```
- (void)removeObjectAtIndex:(unsigned)index; {
   if (index <[embeddedArray count]) {
      [embeddedArray removeObjectAtIndex:index];
   }
}

@end

int main() {
   NSAutoreleasePool * pool = [[NSAutoreleasePool
alloc] init];
   ValidatingArray *validatingArray = [ValidatingArray
validatingArray];

   [validatingArray addObject:@"Object1"];
   [validatingArray addObject:@"Object2"];
   [validatingArray addObject:[NSNull null]];

   [validatingArray removeObjectAtIndex:2];
   NSString *aString = [validatingArray
objectAtIndex:1];
   NSLog(@"Value at the Index 1 is %@",aString);
   [pool drain];

   return 0;
}
```

In the above example, adding null objects to the validating array's one function would result in a crash in a typical circumstance. However, our validating array handles it. Similarly, each method in the validating array adds validating procedures in addition to the standard sequence of operations.

FOUNDATION FRAMEWORK IN OBJECTIVE-C

The specifics of the Foundation framework may be found in Apple documentation, as shown below.

The Foundation framework defines a fundamental layer of Objective-C classes. It includes various paradigms that specify functionality not covered by the Objective-C language and gives a collection of helpful basic object types. The Foundation structure provides these objectives.

- Provide a minimal selection of fundamental utility classes.
- Introduce standard protocols for things like de-allocation to make software development more straightforward.

- Unicode strings, object persistence, and object dissemination are all supported.
- To improve portability, provide some OS independence.

The framework was created by NeXTStep, which Apple later bought, and the foundation classes are included in Mac OS X and iOS. It bears the class prefix "NS" since NeXTStep created it.

All of our example applications have utilized Foundation Framework. Foundation Framework is practically a need.

To import an Objective-C class, we usually use #import Foundation/NSString.h>, but it's all done in #import Foundation/Foundation.h> to prevent importing too many classes.

All objects, including the foundation kit classes, are based on NSObject. It includes memory management techniques. It also has a basic runtime system interface and the ability to act like Objective-C objects. It is the root of all classes and has no base classes.

Functionality-Based Foundation Classes

Sr. No.	Loop Type and Description
1	Data storage NSArray, NSDictionary, and NSSet provide storage for the Objective-C objects of any type.
2	Text and strings The NSCharacterSet class represents multiple character groups utilized by the NSString and NSScanner classes. Text strings are represented by the NSString classes, including methods for finding, combining, and comparing texts. An NSScanner object scans numbers and words from an NSString object.
3	Dates and times The NSDate, NSTimeZone, and NSCalendar classes hold and represent time and date information. They provide ways for determining date and time variations. They provide methods for displaying the dates and times in a variety of formats and altering times and dates based on location throughout the world, in collaboration with NSLocale.
4	Exception handling Exception handling is used to deal with unforeseen occurrences, and it is available in Objective-C via NSException.
5	File handling The class NSFileManager is used to manage files.
6	URL loading system A collection of classes and protocols allows users to access standard Internet protocols.

FAST ENUMERATION IN OBJECTIVE-C

Fast enumeration is a feature of Objective-C that aids in enumerating through a collection. So, to understand quick enumeration, we must first understand the collection covered in the following part.

COLLECTIONS IN THE OBJECTIVE-C

Collections are basic constructs. Its function is to hold and handle other items. The fundamental objective of a collection is to provide a standard mechanism to store and retrieve things effectively.

There are several sorts of collections. While they all provide the same function of being able to contain other items, they primarily differ in how objects are retrieved. The most often used collections in Objective-C are:

- NSArray
- NSSet
- NSMutableSet
- NSDictionary
- NSMutableDictionary
- NSMutableArray

The syntax for fast enumeration

```
for (class_Type variable in collection_Object ) {
   statements
 }
```

Here's an example of a quick enumeration

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

int main() {
   NSAutoreleasePool * pool = [[NSAutoreleasePool
alloc] init];
   NSArray *array = [[NSArray alloc]
   initWithObjects:@"string1",
@"string2",@"string3",nil];
```

```
    for(NSString *aString in array) {
        NSLog(@"The Value is: %@",aString);
    }

    [pool drain];
    return 0;
}
```

Fast enumeration backwards

```
for (class_Type variable in [collection_Object
reverseObjectEnumerator] ) {
    statements
 }
```

In fast enumeration, here's an example of reverseObjectEnumerator.

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

int main() {
    NSAutoreleasePool * pool = [[NSAutoreleasePool
alloc] init];
    NSArray *array = [[NSArray alloc]
    initWithObjects:@"string1",
@"string2",@"string3",nil];

    for(NSString *aString in [array
reverseObjectEnumerator]) {
        NSLog(@"The Value is: %@",aString);
    }

    [pool drain];
    return 0;
}
```

MEMORY MANAGEMENT IN OBJECTIVE-C

Memory management is a necessary procedure in every programming language. It is the process by which objects' memory is allocated when needed and deallocated when they are no longer needed.

Object memory management is a performance issue; if an application does not release unused objects, its memory footprint expands, and performance decreases.

Objective-C Memory management approaches may be divided into two categories.

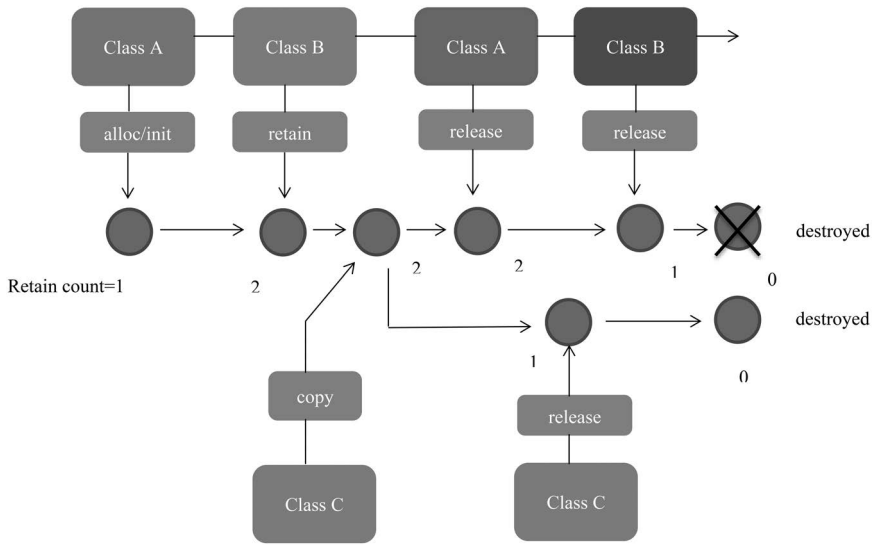
- "Manual Retain-Release" or MRR
- "Automatic Reference Counting" or ARC

"MANUAL RETAIN-RELEASE" OR MRR

In MRR, we manage memory explicitly by keeping track of the items independently. This is accomplished by using a model known as reference counting, which is provided by the Foundation class NSObject in combination with the runtime environment.

The only difference between MRR and ARC in Objective-C is that we handle the retain and release manually in the former while handled automatically in the latter.

The diagram shown below is an example of how memory management works in Objective-C.



Manual retain-release.

The Class A object's memory life cycle is shown in the diagram above. As you can see, the retain count in Objective-C is shown under the object; when the retain count reaches 0, the item is liberated, and its memory is reallocated for other objects to utilize.

The alloc/init function in NSObject is used to construct the class A object. The number of retains is now 1.

Class B now keeps Class A's object, and Class A's object now has a retain count of two.

The item is then copied by Class C. It is now constructed as a new instance of Class A, with the same instance variables. The retain count is 1 instead of the original object's retain count. The dotted line in the diagram represents this.

Class C uses the release method to release the copied object, which causes the keep count to drop to zero and the item to be destroyed.

The keep count for the original Class A Object is two, and it must be released twice before it may be destroyed. This is accomplished using Class A and Class B release statements to set the keep count to 1 and 0, respectively. The thing is finally destroyed.

Basic MRR Rules

- We own whatever we make: A method that starts with "alloc," "new," "copy," or "mutableCopy" is used to construct an object.
- We may use retained to acquire ownership of an object: A received object is usually guaranteed to stay valid inside the method in which it was received. That method may also return the object to its invoker securely. Retain is used in two scenarios.
    - We use an accessor method or an init method to gain ownership of an object we wish to save as a property value.
    - To avoid the invalidation of an object as a result of another action.
- When we no longer need something, we must give up ownership of it: A release message or an autorelease message is used to relinquish control of an item. Consequently, relinquishing ownership of an item is referred to as "releasing" an object in Cocoa parlance.
- We must not give up ownership of something we don't own: This is a consequence of the previously mentioned policy norms.

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

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

```
@implementation SampleClass
- (void)sampleMethod {
   NSLog(@"Hello, Everyone \n");
}

- (void)dealloc  {
  NSLog(@"Object deallocated");
  [super dealloc];
}

@end

int main() {

   /* my first program in the Objective-C */
   SampleClass *sampleClass = [[SampleClass alloc]
init];
   [sampleClass sampleMethod];

   NSLog(@"Retain Count after the initial allocation:
%d",
   [sampleClass retainCount]);
   [sampleClass retain];

   NSLog(@"Retain Count after the retain: %d",
[sampleClass retainCount]);
   [sampleClass release];
   NSLog(@"Retain Count after the  release: %d",
[sampleClass retainCount]);
   [sampleClass release];
   NSLog(@"SampleClass dealloc will call before
this");

   // Should set object to nil
   sampleClass = nil;
   return 0;
}
```

"AUTOMATIC REFERENCE COUNTING" OR ARC

Automatic Reference Counting, or ARC, employs the same reference counting approach as MRR, but it inserts the proper memory management method calls for us at build time. For new projects, we are highly recommended to adopt ARC. If we utilize ARC, we usually don't need to know about the underlying implementation detailed in this chapter, while

it could be helpful in some instances. See Transitioning to ARC Release Notes for further information on ARC.

We do not need to introduce release and retain methods in ARC since the compiler will take care of it. Objective-core C's methodology hasn't changed. Internally, it makes it simpler for the developer to write without worrying about the retain and release actions, reducing the amount of code written and the risk of memory leaks.

Another notion called garbage collection is utilized in Mac OS X alongside MRR, but it hasn't been acknowledged since its deprecation in OS X Mountain Lion. Furthermore, trash collection was never available for iOS objects. Garbage collection is also not used on OS X while using ARC.

Here's an example of an ARC. Because the online compiler does not support ARC, this will not work.

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

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

@implementation SampleClass
- (void)sampleMethod {
   NSLog(@"Hello, everyone \n");
}

- (void)dealloc  {
  NSLog(@"Object deallocated");
}

@end

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

In this chapter, we covered OOP's concept with its relevant examples.

BIBLIOGRAPHY

1. Object-Oriented Programming with Objective-C.
2. Obj-C Memory Management.
3. Objective-C Environment Setup – https://www.tutorialspoint.com/objective_c/objective_c_environment_setup.htm#:~:text=In%20order%20to%20run%20Objective,install%20the%20GNUstep%20Core%20package, accessed on May 6, 2022.
4. Installing compiler for Objective-C – http://referencedesigner.com/tutorials/objectivec/objectivec_02.php, accessed on May 6, 2022.
5. 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 May 6, 2022.
6. Objective-C Program Structure – https://www.tutorialspoint.com/objective_c/objective_c_program_structure.htm, accessed on May 7, 2022.
7. Structures in Objective-C – https://www.educative.io/answers/what-are-structures-in-objective-c, accessed on May 7, 2022.
8. Objective-C Code – https://developer.apple.com/library/archive/referencelibrary/GettingStarted/RoadMapiOS-Legacy/chapters/WriteObjective-CCode/WriteObjective-CCode/WriteObjective-CCode.html, accessed on May 7, 2022.
9. Installing Xcode and Compiling Objective-C on Mac OS X – https://www.techotopia.com/index.php/Installing_Xcode_and_Compiling_Objective-C_on_Mac_OS_X#Installing_Xcode_on_Mac_OS_X, accessed on May 7, 2022.
10. Classes Are Blueprints for Objects – https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/DefiningClasses/DefiningClasses.html#:~:text=Working%20with%20Objects.,Objective%2DC%20Classes%20Are%20also%20Objects,but%20they%20can%20receive%20messages, accessed on May 7, 2022.
11. Objective-C Classes & Objects – https://www.tutorialspoint.com/objective_c/objective_c_classes_objects.htm, accessed on May 9, 2022.
12. Quick Refresh: What Is Objective-C – https://blog.teamtreehouse.com/beginners-guide-objective-c-classes-objects, accessed on May 9, 2022.
13. Classes, Objects, and Methods in Objective-C – https://www.informit.com/articles/article.aspx?p=1722550&seqNum=7, accessed on May 9, 2022.
14. Objective-C Instances – http://www.apeth.com/iOSBook/ch05.html, accessed on May 9, 2022.

CHAPTER 3

Interface and API

IN THIS CHAPTER

- Interfaces
- APIs

In the previous chapter, we discussed OOP's concept in Objective-C, and in this chapter, we will cover Interface and API.

iOS IN OBJECTIVE-C

Objective-C is the programming language used in iOS development. It is an object-oriented language. Thus, individuals with experience with object-oriented programming languages will find it simple.

IMPLEMENTATION AND INTERFACE

In Objective-C, the file where the class is declared is referred to as the interface file, and the file where the class is defined as the implementation file.

A basic interface file, MyExample.h, might look like this:

```
@interface MyClass:NSObject {
   // class-variable declared
}

// class-properties declared
// class-methods and instance-methods declared
@end
```

DOI: 10.1201/9781003357506-3

The implementation file MyExample.m might look like this:

```
@implementation MyClass
   // class-methods defined
@end
```

OBJECT CREATION

The following is how objects are created:

```
MyExample *objectName = [[MyClass alloc]init] ;
```

METHODS

The following method is declared in Objective-C:

```
-(returnType)method_Name:(type_Name) variable1 :(type_
Name)variable2;
```

Example:

```
-(void)calculateAreaForRectangleWithLength:(CGfl
oat)length
andBreadth:(CGfloat)breadth;
```

We may be wondering what the andBreadth string is for; it's an optional string that helps us read and comprehend the method more readily, especially when we call it. We use the following line to call this function in the same class:

```
[self calculateAreaForRectangleWithLength:40
andBreadth:30];
```

As previously stated, the usage of andBreadth aids us in understanding that breadth is 20. Self is used to indicate that the method is a class method.

Class Methods

Class methods are accessed without first constructing a class object. They are not linked with any variables or objects. An example is shown below.

```
+(void)simpleClass_Method;
```

These are accessed by using the class name (let's say MyExample) as follows:

```
[MyExample simpleClass_Method];
```

Instance Methods

Instance methods are accessed only once a class object has been created. Memory is set aside for instance variables. The following is an example of an instance method.

```
-(void)simpleInstance_Method;
```

It may be accessed after creating a class object as follows:

```
MyExample  *objectName = [[MyExample alloc]init] ;
[object_Name simpleInstance_Method];
```

IMPORTANT OBJECTIVE-C DATA TYPES

Sr. No.	Data Type
1	NSString It is a string representation.
2	CGfloat It represents a floating-point value (normal float is also allowed, but CGfloat is preferred).
3	NSInteger It's used to represent integers.
4	BOOL It is used to express Boolean operations (YES or NO are BOOL types allowed).

Printing Logs

NSLog – This function is used to print a statement. In the release and debug modes, they will be displayed in the device logs and debug console, respectively. As an example,

```
NSlog(@"");
```

CONTROL STRUCTURES

Except for a few modifications, such as the for-in statement, most control structures are the same as in C and C++.

PROPERTIES

Variable properties are used to allow an external class to access the class. As an example,

```
@property(nonatomic,  strong) NSString *myString;
```

Properties of Accessing

To access properties, use the dot operator. We will do the following to access the property, as mentioned earlier.

```
self.myString = @"Test";
```

We may also use the set approach, as seen below.

```
[self setMyString:@"Test"];
```

CATEGORIES

Categories are used to extend existing classes with new methods. We may add methods to classes that don't even have implementation files where the real class is defined. The following is an example category for our class:

```
@interface MyExample(customAdditions)
- (void)sampleCategoryMethod;
@end

@implementation MyExample(categoryAdditions)

-(void)sampleCategoryMethod {
   NSLog(@"Just test category");
}
```

Arrays

The array classes used in Objective-C are NSMutableArray and NSArray. The name implies that the former is mutable, whereas the latter is immutable. An example is shown below.

```
NSMutableArray *bMutableArray = [[NSMutableArray
alloc]init];
[anArray addObject:@"firstobject"];
NSArray *bImmutableArray = [[NSArray alloc]
initWithObjects:@"firstObject",nil];
```

Dictionary

The dictionary classes used in Objective-C are NSMutableDictionary and NSDictionary. The name implies that the former is mutable, whereas the latter is immutable. An example is shown below.

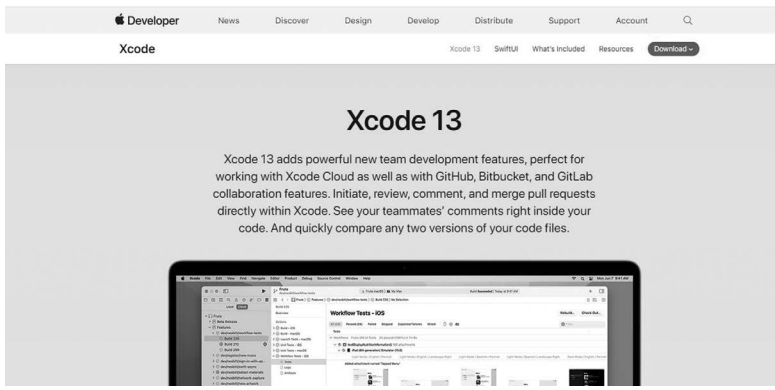
```
NSMutableDictionary *bMutableDictionary =
[[NSMutableArray alloc]init];
[bMutableDictionary setObject:@"firstobject"
forKey:@"aKey"];
NSDictionary*bImmutableDictionary= [[NSDictionary
alloc]initWithObjects:[NSArray arrayWithObjects:
@"firstObject",nil] forKeys:[ NSArray
arrayWithObjects:@"bKey"]];
```

ENVIRONMENT SETUP

Here we will discuss the environment setup.

Installation of Xcode

- Step 1: Download the most recent version of Xcode from: https://developer.apple.com/downloads/.



- Step 2: Open the Xcode dmg file with a double-click.
- Step 3: We will discover a gadget that has been installed and opened.
- Step 4: The window will show two items: the Xcode program and the shortcut to the Application folder.
- Step 5: Drag the Xcode to the application to copy it to our apps.

- Step 6: Xcode will now be available as part of other apps that we may pick and execute.

INTERFACE BUILDER

An interface builder is a tool that allows for the quick construction of UI interfaces. We have a diverse range of UI components that have been designed for usage. We just drag and drop it into our UI view. We'll learn about adding UI elements, generating outlets, and creating actions for the UI elements in the following pages.

At the right bottom, an object library contains all the essential UI elements. The file extension xibs is commonly used to refer to the user interface. Each of the xibs is associated with a view controller.

SIMULATOR FOR iOS

An iOS emulator comprises two sorts of devices: iPhones and iPads in various versions. iPhone models include the iPhone (standard), iPhone Retina, and iPhone 5. The iPad comes in two varieties: iPad and iPad Retina.

We may replicate the location in an iOS simulator to experiment with the app's latitude and longitude effects. In the simulator, we can also mimic memory warnings and in-call status. The simulator can be used for most purposes, although it cannot test device functions such as the accelerometer. As a result, we may always require an iOS device to test all application situations thoroughly.

FIRST iPHONE APPLICATION

Developing the First App:

We'll make a simple single-view application (a blank App) to run on the iOS simulator.

The procedures are as follows.

- Step 1: Launch Xcode and choose to Create a new Xcode project.
- Step 2: Click on Single View Application.
- Step 3: Enter the product name, the application name, the organization name, and finally, the corporate identification.
- Step 4: Ensure that Use Automatic Reference Counting is enabled to automatically release the resources allocated when they are no longer needed. Next, click.

- Step 5: Choose the project directory and click Create.
- Step 6: We will be able to choose between supported orientations and build and release options. There is a field deployment goal: the device version we want to support; let's say we choose 4.3, the minimum deployment target allowed right now. These are not necessary, and we may concentrate on executing the program for the time being.
- Step 7: Now, pick the iPhone simulator from the drop-down beside the Run button and press the Run button.
- Step 8: We have now successfully run our first application.

Let's modify the backdrop color to get started with the interface builder. Choose ViewController.xib. Change the color of the backdrop on the right side and run.

The deployment target is configured to iOS 6.0 by default, and auto-layout is enabled. We adjusted the deployment target at the start of this application's construction to ensure it operates on iOS 4.3 and later devices, but we did not disable auto-layout.

To turn off auto-layout, uncheck the auto-layout checkbox in the file inspector of each nib, that is, the xib files. The parts of the Xcode project IDE are listed.

The file inspector is located in the inspector selector bar, and auto-layout may be unchecked there. We may utilize auto-layout when we want to target iOS 6 devices. We will also be able to use several new features, such as a passbook if we increase the deployment target to iOS 6. For the time being, let's remain with iOS 4.3 as the deployment target.

FIRST iOS APPLICATION'S CODE

Five separate files were created for your application. The following is a list of them:

- AppDelegate.h
- AppDelegate.m
- ViewController.h
- ViewController.m
- ViewController.xib

AppDelegate.h

```
// Header File that provides all the UI related items.
#import <UIKit/UIKit.h>

// Forward-declaration (Used when class will define /
imported in the future)
@class ViewController;

// Interface for the Appdelegate
@interface AppDelegate : UIResponder
<UIApplicationDelegate>

// Property-window
@property (strong, nonatomic) UIWindow *window;

// Property-Viewcontroller

@property (strong, nonatomic) ViewController
*viewController;
//this marks end of the interface
@end
```

Important code elements:

- AppDelegate is a subclass of UIResponder, which handles iOS events.
- Implements the UIApplicationDelegate delegate methods, which give critical application events such as finished launching, about to terminate, etc.
- To manage and coordinate the multiple views on the iOS device screen, use the UIWindow object. It's similar to the base view on top of which all additional views are loaded. An application typically has only one window.
- UIViewController handles the screen flow.

AppDelegate.m

```
// Imports class Appdelegate's interface
import "AppDelegate.h"
```

```
// Imports viewcontroller to load
#import "ViewController.h"

// Here Class definition starts
@implementation AppDelegate

// Method to intimate us that application launched
successfully
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)
launchOptions {
   self.window = [[UIWindow alloc]
initWithFrame:[[UIScreen mainScreen] bounds]];

   // Override point for the customization after the
application launch.
   self.viewController = [[ViewController alloc]
   initWithNibName:@"ViewController" bundle:nil];
   self.window.rootViewController = self.
viewController;
   [self.window makeKeyAndVisible];
   return YES;
}

- (void)applicationWillResignActive:(UIApplication *)
application {
   /* Use this method to release the shared resources,
save user data,
   invalidate timers, and store enough application
state information
   to restore our application to its current state if
it is
   terminated later. If our application supports
background
   execution, this method is called instead of
   applicationWillTerminate: when a user quits.*/
}

- (void)applicationWillEnterForeground:(UIApplication
*)application {
   /* Called as part of transition from the background
to the
```

```
   inactive state. Here we can undo many of the
changes made on
   entering background.*/
}

- (void)applicationDidBecomeActive:(UIApplication *)
application {
   /* Restart any tasks that were paused (or not yet
started) while
   application was inactive. If the application was
previously in
   background, optionally refresh user interface.*/
}

- (void)applicationWillTerminate:(UIApplication *)
application {
   /* Called when application is about to terminate.
Save data if
   appropriate. See also
applicationDidEnterBackground:. */
}

- (void)applicationWillTerminate:(UIApplication *)
application {
   /* Called when application is about to terminate.
Save data if appropriate.
   See also applicationDidEnterBackground:. */
}
@end
```

Important code elements:

- Delegates for UIApplications are specified here. The methods specified here are UI application delegates with no user-defined methods.
- A UIWindow object is created to hold the program.
- As the window's first view controller, UIViewController is assigned.
- The makeKeyAndVisible function is used to make the window visible.

ViewController.h

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

// Interface for the class ViewController
@interface ViewController : UIViewController

@end
```

Important code elements:

- The ViewController class derives from the UIViewController class, which provides the primary view management mechanism for iOS apps.

ViewController.m

```
#import "ViewController.h"

// Category, extension of the ViewController class
@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad {
   [super viewDidLoad];
   // Do any additional setup after the loading view,
typically from nib.
}

- (void)didReceiveMemoryWarning {
   [super didReceiveMemoryWarning];
   // Dispose of any resources that can recreate.
}
@end
```

Important code elements:

- The introductory class UIViewController defines two methods that are used here.

- Perform initial setup in viewDidLoad, which is called after the view has loaded.
- In the case of a memory warning, the didReceiveMemoryWarning method is invoked.

ACTIONS AND OUTLETS IN iOS

In iOS, actions and outlets are called ibActions and ibOutlets, respectively, where ib stands for interface builder. These are UI components, and we will investigate them after we have a visual understanding of how to apply them.

Actions and Outlets – Involved Steps:

- Step 1: Let's utilize our First iPhone Application.
- Step 2: Choose the ViewController.xib file from the navigation section.
- Step 3: We can now choose UI components from the library pane on the right side of our window.
- Step 4: In our interface builder, drag and drop the UI elements to our display.
- Step 5: Insert a Label and a Round Rect Button into our view.
- Step 6: Click the Editor Selector button in the workspace toolbar, located in the upper right corner.

Select the Assistant editor button.

- Step 7: In the center of our editor area, we notice two windows: ViewController.xib and ViewController.h.
- Step 8: Right-click the label and select, hold, and drag the new referring outlet.
- Step 9: Insert the ViewController. h in the space between the curly brackets. If there are no curly brackets in the file, add the ViewController first.
- Step 10: Enter the label name for the outlet, in this case, myTitleLabel. When you click connect, the ibOutlet will finish.

- Step 11: To add an action, right-click the Round rect button, choose Touch up inside, and drag it below the curly braces.
- Step 12: Drop it and give it the name setTitleLabel.
- Step 13: Open the ViewController.m file and look for the function described below.

```
-(IBAction) setTitleLabel:(id)sender {
}
```

- Step 14: Insert the following statement within the procedure mentioned above.

```
[myTitleLabel setText:@"Heyyy"];
```

- Step 15: Now, start the application by clicking the run button.
- Step 16: Now press the button.
- Step 17: The action on the button has updated the label that we established.
- Step 18: From the above example, IBOutlet establishes a reference to the UIElement (here for the UILabel). Similarly, the IBAction associates the UIButton with a method invoked at the event touch-up inside.
- Step 19: We may experiment with actions by selecting different events as we create them.

DELEGATES IN iOS

Delegation example:

Assume that object A requests an action from object B. Object A should be aware that B has completed the task and take the appropriate action when the activity is finished. This is accomplished with the assistance of delegates.

The main principles in the preceding example are:

- B's delegate object is A.
- A will be a reference for B.
- B's delegate methods will be implemented by A.
- B will inform A via delegate techniques.

How to Create a Delegate

- Step 1: Create a single view application first.
- Step 2: Then choose File → New → File.
- Step 3: Select Objective-C Class and then click Next.
- Step 4: Name the class SampleProtocols, with NSObject as a subclass.
- Step 5: Then click Create.
- Step 6: Add a protocol to the SampleProtocols.h file, and the updated code looks like this:

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

// Here Protocol definition starts
@protocol SampleProtocolsDelegate <NSObject>
@required
- (void) processCompleted;
@end

// Here Protocol Definition ends
@interface SampleProtocols : NSObject {
   // Delegate to respond-back
   id <SampleProtocolsDelegate> _delegate;
}
@property (nonatomic,strong) id delegate;

-(void)startSampleProcess; // Instance-method
@end
```

- Step 7: Put the instance method into action by changing the SampleProtocols.m file below.

```
#import "SampleProtocols.h"

@implementation SampleProtocols

-(void)startSampleProcess {
   [NSTimer scheduledTimerWithTimeInterval:3.0
target:self.delegate
   selector:@selector(processCompleted)
userInfo:nil repeats:NO];
}
@end
```

- Step 8: Drag a UILabel from the object library onto the UIView in ViewController.xib.
- Step 9: Create an IBOutlet for the label, call it myLabel, and then alter the code in ViewController.h to use SampleProtocolsDelegate.

```
#import <UIKit/UIKit.h>
#import "SampleProtocols.h"

@interface ViewController : UIViewController<Sampl
eProtocolDelegate> {
   IBOutlet UILabel *myLabel;
}
@end
```

- Step 10: Implement the delegate method, create a SampleProtocols object, and invoke the startSampleProcess method. The updated ViewController.m file looks like this:

```
#import "ViewController.h"

@interface ViewController ()

@end

@implementation ViewController

- (void)viewDidLoad {
   [super viewDidLoad];
   SampleProtocol *sampleProtocols =
[[SampleProtocols alloc]init];
   sampleProtocol.delegate = self;
   [myLabel setText:@"Processing...."];
   [sampleProtocols startSampleProcess];
   // Do any additional setup after loading view,
typically from nib.
}

- (void)didReceiveMemoryWarning {
   [super didReceiveMemoryWarning];
   // Dispose of any resources that can recreate.
}
```

```
#pragma mark - Sample protocols delegate
-(void)processCompleted {
    [myLabel setText:@"Process-Completed"];
}
@end
```

- Step 11: We will see the result. The label initially shows "processing...," which is changed when the SampleProtocols object calls the delegate method.

UI ELEMENTS

What Are UI Elements?

UI elements are the visual components of an application's user interface. Some of these components are interactive, like buttons and text boxes, while others are instructive, like photos and labels.

How Do We Insert UI Elements?

We can add UI components using both code and an interface builder. Depending on the situation, we can use either option.

Our Focus

In our apps, we'll emphasize adding UI components using code. The interface builder is basic and straightforward; UI components are simply dragged and dropped.

Our Strategy

We will develop a basic iOS application and use it to demonstrate some UI features.

- Step 1: Create a Viewbased application, much like we did with our first iOS app.
- Step 2: Only the ViewController.h and ViewController.m files will be updated.
- Step 3: In our ViewController.m file, we add a function for constructing the UI element.
- Step 4: In our viewDidLoad function, we will call this method.
- Step 5: The key lines of code in the code have been clarified using a single-line comment above those lines.

LIST OF UI ELEMENTS

The following table lists the UI components and their related capabilities:

Sr. No.	UI Specific Elements
1	Text Fields A user interface element allows the program to receive user input.
2	Input types – TextFields Using the keyboard property of UITextField, we can specify the type of input that the user can provide.
3	Buttons It is responsible for managing user actions.
4	Label It's used to show static content.
5	Toolbar It is utilized to modify anything based on our present point of view.
6	Status Bar It displays the device's vital information.
7	Navigation Bar It has navigation buttons from a navigation controller and a stack of view controllers that can be pushed and popped.
8	Tab Bar It is typically used to switch between different subtasks, views, or models inside the same view.
9	Image View It's used to show a single image or a series of images.
10	Scroll View It is used to display larger content than the screen's size.
11	Table View It shows a scrollable list of data in numerous rows and sections.
12	Split View It is used to show two panes, with the master pane controlling the information on the detail pane.
13	Text View It displays a scrollable list of text content that may be edited if needed.
14	View Transition It describes the different view transitions between perspectives.
15	Pickers It is used for showing and choosing data from a list.
16	Switches It serves as a disable and enables switch for activities.
17	Sliders It allows users to make changes to a value or process across a range of possible values.
18	Alerts It is used to provide consumers with critical information.
19	Icons It is a visual depiction of the activity or anything linked to the program.

ACCELEROMETER IN iOS

The accelerometer detects changes in the device's location in the three directions x, y, and z. We can determine the device's current position concerning the ground. This example requires running on a device and does not function in a simulator.

Steps Involved with an accelerometer:

- Step 1: Make a basic View-based application.
- Step 2: In ViewController, add three labels.
  xib and make ibOutlets named xlabel, ylabel, and zlabel.
- Step 3: Make the following changes to ViewController.h:

```
#import <UIKit/UIKit.h>
@interface ViewController : UIViewController<UIAcc
elerometerDelegate> {
   IBOutlet UILabel *xlabel;
   IBOutlet UILabel *ylabel;
   IBOutlet UILabel *zlabel;
}
@end
```

- Step 4: Make the following changes to ViewController.m:

```
#import "ViewController.h"

@interface ViewController ()
@end

@implementation ViewController

- (void)viewDidLoad {
   [super viewDidLoad];
   [[UIAccelerometer sharedAccelerometer]
setDelegate:self];
   //Do any additional setup after loading
view,typically from nib
}

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

```
    // Dispose of any resources that can recreate.
}

- (void)accelerometer:(UIAccelerometer *)
accelerometer didAccelerate:
    (UIAcceleration *)acceleration {
    [xlabel setText:[NSString stringWithFormat:@"%f
",acceleration.x]];
    [ylabel setText:[NSString stringWithFormat:@"%f
",acceleration.y]];
    [zlabel setText:[NSString stringWithFormat:@"%f
",acceleration.z]];
}
@end
```

UNIVERSAL APPLICATIONS IN iOS

A universal application is created in a single binary for both the iPhone and the iPad. A universal App enables code reuse and quick upgrades.

Steps involved in universal application

- Step 1: Make a basic View-based application.
- Step 2: In the right-hand file inspector, rename the ViewController.xib file to ViewController iPhone.xib.
- Step 3: Select File → New → File... then the subheading "User Interface" and View. Next, click.
- Step 4: Select iPad as the device family and then click Next.
- Step 5: Select Create and save the file as ViewController iPad.xib.
- Step 6: In both ViewController iPhone.xib and ViewController iPad.xib, place a label in the center of the screen.
- Step 7: Select the identity inspector in ViewController iPad.xib and change the custom class to ViewController.
- Step 8: Update the application: In AppDelegate.m, use the DidFinishLaunching:withOptions function as follows:

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)
launchOptions {
```

```
    self.window = [[UIWindow alloc]
initWithFrame:[[UIScreen
    mainScreen] bounds]];

    // Override point for the customization after
the application launch.
    if (UI_USER_INTERFACE_IDIOM() ==
UIUserInterfaceIdiomPhone) {
        self.viewController = [[ViewController
alloc]
        initWithNibName:@"ViewController_iPhone"
bundle:nil];
    } else {
        self.viewController = [[ViewController
alloc] initWithNibName:
        @"ViewController_iPad" bundle:nil];
    }
    self.window.rootViewController = self.
viewController;
    [self.window makeKeyAndVisible];
    return YES;
}
```

- Step 9: Change the device in the project summary to universal.

CAMERA MANAGEMENT IN iOS

A camera is a typical feature of a mobile device. We may snap pictures using the camera and utilize them in our program; it's also reasonably straightforward.

Steps involved in camera management

- Step 1: Make a basic View-based application.
- Step 2: Add a button to ViewController.xib and build an IBAction for it.
- Step 3: Add an image view and build an IBOutlet called imageView.
- Step 4: Make the following changes to ViewController.h:

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

@interface ViewController : UIViewController<UIIma
gePickerControllerDelegate> {
```

```
    UIImagePickerController *imagePicker;
    IBOutlet UIImageView *imageView;
}

- (IBAction)showCamera:(id)sender;
@end
```

- Step 5: Make the following changes to ViewController.m:

```
#import "ViewController.h"

@interface ViewController ()
@end

@implementation ViewController

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

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can recreate.
}

- (IBAction)showCamera:(id)sender {
    imagePicker.allowsEditing = YES;

    if ([UIImagePickerController
isSourceTypeAvailable:
    UIImagePickerControllerSourceTypeCamera]) {
        imagePicker.sourceType =
UIImagePickerControllerSourceTypeCamera;
    } else {
        imagePicker.sourceType =

UIImagePickerControllerSourceTypePhotoLibrary;
    }
    [self presentModalViewController:imagePicker
animated:YES];
}

-(void)imagePickerController:(UIImagePickerControl
ler *)picker
```

```
   didFinishPickingMediaWithInfo:(NSDictionary *)info
{
      UIImage *image = [info objectForKey:UIImagePicke
rControllerEditedImage];

      if (image == nil) {
         image = [info objectForKey:UIImagePickerContr
ollerOriginalImage];
      }
   imageView.image = image;
}

-(void)imagePickerControllerDidCancel:(UIImagePickerCo
ntroller *)picker {
   [self dismissModalViewControllerAnimated:YES];
}
@end
```

LOCATION HANDLING IN iOS

We can simply find the user's current position in iOS if the user allows the application access to the information using the core location framework.

Steps involved in location handling

- Step 1: Make a basic View-based application.
- Step 2: Choose our project file, then targets, and add CoreLocation. framework.
- Step 3: In ViewController, add two labels. xib and build ibOutlets with the names latitudeLabel and longitudeLabel.
- Step 4: Select File → New → File... → select Objective-C class and click next.
- Step 5: Give the class the name LocationHandler and the "subclass of" as NSObject.
- Step 6: Click the Create button.
- Step 7: Make the following changes to LocationHandler.h:

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

@protocol LocationHandlerDelegate <NSObject>
```

```
@required
-(void) didUpdateToLocation:(CLLocation*)
newLocation
   fromLocation:(CLLocation*)oldLocation;
@end

@interface LocationHandler : NSObject<CLLocationMa
nagerDelegate> {
   CLLocationManager *locationManager;
}
@property(nonatomic,strong)
id<LocationHandlerDelegate> delegate;

+(id)getSharedInstance;
-(void)startUpdating;
-(void) stopUpdating;

@end
```

- Step 8: Make the following changes to LocationHandler.m:

```
#import "LocationHandler.h"
static LocationHandler *DefaultManager = nil;

@interface LocationHandler()

-(void)initiate;

@end
//implementation
@implementation LocationHandler

+(id)getSharedInstance{
   if (!DefaultManager) {
      DefaultManager = [[self allocWithZone:NULL]
init];
       [DefaultManager initiate];
   }
   return DefaultManager;
}

-(void)initiate {
   locationManager = [[CLLocationManager alloc]
init];
```

```
    locationManager.delegate = self;
}

-(void)startUpdating{
    [locationManager startUpdatingLocation];
}

-(void) stopUpdating {
    [locationManager stopUpdatingLocation];
}

-(void)locationManager:(CLLocationManager *)
manager didUpdateToLocation:
    (CLLocation *)newLocation
fromLocation:(CLLocation *)oldLocation {
    if ([self.delegate respondsToSelector:@selector
    (didUpdateToLocation:fromLocation:)]) {
        [self.delegate
didUpdateToLocation:oldLocation
        fromLocation:newLocation];
    }
}
@end
```

- Step 9: Update ViewController.h to include the LocationHandler delegate and two ibOutlets.

```
#import <UIKit/UIKit.h>
#import "LocationHandler.h"

@interface ViewController : UIViewController<Locat
ionHandlerDelegate> {
    IBOutlet UILabel *latitudeLabel;
    IBOutlet UILabel *longitudeLabel;
}
@end
```

- Step 10: Make the following changes to ViewController.m:

```
#import "ViewController.h"

@interface ViewController ()
@end
```

```
@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    [[LocationHandler getSharedInstance]
setDelegate:self];
    [[LocationHandler getSharedInstance]
startUpdating];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can recreate.
}

-(void)didUpdateToLocation:(CLLocation *)
newLocation
 fromLocation:(CLLocation *)oldLocation {
    [latitudeLabel setText:[NSString
stringWithFormat:
   @"Latitude: %f",newLocation.coordinate.
latitude]];
    [longitudeLabel setText:[NSString
stringWithFormat:
    @"Longitude: %f",newLocation.coordinate.
longitude]];
}
@end
```

SQLite DATABASE IN iOS

SQLite is used to manage data on iOS. It employs sqlite queries, making it simpler for individuals familiar with SQL.

Steps required

- Step 1: Make a basic View-based application.
- Step 2: Select your project file, then targets, and finally the libsqlite3.dylib library in frameworks.
- Step 3: Create a new file by choosing File New File..., then selecting Objective-C class and clicking Next.

- Step 4: Name the class DBManager and include "sub class of" as NSObject.
- Step 5: Click Create.
- Step 6: Make the following changes to DBManager.h:

```
#import <Foundation/Foundation.h>
#import <sqlite3.h>

@interface DBManager : NSObject {
   NSString *databasePath;
}

+(DBManager*)getSharedInstance;
-(BOOL)createDB;
-(BOOL) saveData:(NSString*)registerNumber
name:(NSString*)name
   department:(NSString*)department
year:(NSString*)year;
-(NSArray*) findByRegisterNumber:(NSString*)
registerNumber;

@end
```

- Step 7: Make the following changes to DBManager.m:

```
#import "DBManager.h"
static DBManager *sharedInstance = nil;
static sqlite3 *database = nil;
static sqlite3_stmt *statement = nil;

@implementation DBManager

+(DBManager*)getSharedInstance {
   if (!sharedInstance) {
      sharedInstance = [[super allocWithZone:NULL]
init];
      [sharedInstance createDB];
   }
   return sharedInstance;
}

-(BOOL)createDB {
   NSString *docsDir;
   NSArray *dirPaths;
```

```
   // Get documents directory
   dirPaths = NSSearchPathForDirectoriesInDomains
   (NSDocumentDirectory, NSUserDomainMask, YES);
   docsDir = dirPaths[0];

   // Build path to the database file
   databasePath = [[NSString alloc]
initWithString:
   [docsDir stringByAppendingPathComponent:
@"student.db"]];
   BOOL isSuccess = YES;
   NSFileManager *filemgr = [NSFileManager
defaultManager];

   if ([filemgr fileExistsAtPath: databasePath ]
== NO) {
      const char *dbpath = [databasePath
UTF8String];
      if (sqlite3_open(dbpath, &database) ==
SQLITE_OK) {
         char *errMsg;
         const char *sql_stmt =
         "create the table if not exists
studentsDetail (regno integer
         primary key, name text, department text,
year text)";

         if (sqlite3_exec(database, sql_stmt,
NULL, NULL, &errMsg) != SQLITE_OK) {
            isSuccess = NO;
            NSLog(@"Failed to create the table");
         }
         sqlite3_close(database);
         return  isSuccess;
      } else {
         isSuccess = NO;
         NSLog(@"Failed to open/create the
database");
      }
   }
   return isSuccess;
}
```

```
- (BOOL) saveData:(NSString*)registerNumber
name:(NSString*)name
   department:(NSString*)department
year:(NSString*)year; {
   const char *dbpath = [databasePath UTF8String];

   if (sqlite3_open(dbpath, &database) == SQLITE_
OK) {
      NSString *insertSQL = [NSString
stringWithFormat:@"insert into
      studentsDetail (regno,name, department,
year) values
      (\"%d\",\"%@\", \"%@\",
\"%@\")",[registerNumber integerValue],
      name, department, year];
      const char *insert_stmt = [insertSQL
UTF8String];
      sqlite3_prepare_v2(database, insert_stmt,-1,
&statement, NULL);

      if (sqlite3_step(statement) == SQLITE_DONE)
{
         return YES;
      } else {
         return NO;
      }
      sqlite3_reset(statement);
   }
   return NO;
}

- (NSArray*) findByRegisterNumber:(NSString*)
registerNumber {
   const char *dbpath = [databasePath UTF8String];

   if (sqlite3_open(dbpath, &database) == SQLITE_
OK) {
      NSString *querySQL = [NSString
stringWithFormat:
      @"select name, department, year from
studentsDetail where
      regno=\"%@\"",registerNumber];
```

```
      const char *query_stmt = [querySQL
UTF8String];
      NSMutableArray *resultArray =
[[NSMutableArray alloc]init];

      if (sqlite3_prepare_v2(database, query_stmt,
-1, &statement, NULL) == SQLITE_OK) {
         if (sqlite3_step(statement) == SQLITE_
ROW) {
            NSString *name = [[NSString alloc]
initWithUTF8String:
            (const char *) sqlite3_column_
text(statement, 0)];
            [resultArray addObject:name];

            NSString *department = [[NSString
alloc] initWithUTF8String:
            (const char *) sqlite3_column_
text(statement, 1)];
            [resultArray addObject:department];

            NSString *year = [[NSString alloc]
initWithUTF8String:
            (const char *) sqlite3_column_
text(statement, 2)];
            [resultArray addObject:year];
            return resultArray;
         } else {
            NSLog(@"Notfound");
            return nil;
         }
         sqlite3_reset(statement);
      }
   }
   return nil;
}
```

- Step 8: Update the ViewController.xib file.
- Step 9: Create IBOutlets for the text fields mentioned above.
- Step 10: Make an IBAction for the buttons.

- Step 11: Update ViewController.h as follows:

```
#import <UIKit/UIKit.h>
#import "DBManager.h"

@interface ViewController : UIViewController<UITex
tFieldDelegate> {
   IBOutlet UITextField *regNoTextField;
   IBOutlet UITextField *nameTextField;
   IBOutlet UITextField *departmentTextField;
   IBOutlet UITextField *yearTextField;
   IBOutlet UITextField
*findByRegisterNumberTextField;
   IBOutlet UIScrollView *myScrollView;
}

-(IBAction)saveData:(id)sender;
-(IBAction)findData:(id)sender;
@end
```

- Step 12: Make the following changes to ViewController.m:

```
#import "ViewController.h"

@interface ViewController ()
@end

@implementation ViewController

- (id)initWithNibName:(NSString *)nibNameOrNil
bundle:(NSBundle *)
   nibBundleOrNil {
   self = [super initWithNibName:nibNameOrNil
bundle:nibBundleOrNil];

   if (self) {
      // Custom-initialization
   }
   return self;
}

- (void)viewDidLoad {
   [super viewDidLoad];
   // Do any additional setup after the loading
the view from its nib.
}
```

```
- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can recreate.
}

-(IBAction)saveData:(id)sender {
    BOOL success = NO;
    NSString *alertString = @"Data Insertion
failed";

    if (regNoTextField.text.length>0
&&nameTextField.text.length>0 &&
        departmentTextField.text.length>0
&&yearTextField.text.length>0 ) {
        success = [[DBManager getSharedInstance]
saveData:
        regNoTextField.text name:nameTextField.text
department:
        departmentTextField.text year:yearTextField.
text];
    } else {
        alertString = @"Enter all the fields";
    }

    if (success == NO) {
        UIAlertView *alert = [[UIAlertView alloc]
initWithTitle:
        alertString message:nil
        delegate:nil cancelButtonTitle:@"OK"
otherButtonTitles:nil];
        [alert show];
    }
}

-(IBAction)findData:(id)sender {
    NSArray *data = [[DBManager getSharedInstance]
findByRegisterNumber:
    findByRegisterNumberTextField.text];

    if (data == nil) {
        UIAlertView *alert = [[UIAlertView alloc]
initWithTitle:
        @"Data not found" message:nil delegate:nil
cancelButtonTitle:
        @"OK" otherButtonTitles:nil];
        [alert show];
```

```
        regNoTextField.text = @"";
        nameTextField.text =@"";
        departmentTextField.text = @"";
        yearTextField.text =@"";
    } else {
        regNoTextField.text =
findByRegisterNumberTextField.text;
        nameTextField.text =[data objectAtIndex:0];
        departmentTextField.text = [data
objectAtIndex:1];
        yearTextField.text =[data objectAtIndex:2];
    }
}

#pragma mark - Text field delegate
-(void)textFieldDidBeginEditing:(UITextField *)
textField {
    [myScrollView setFrame:CGRectMake(20, 60, 400,
300)];
    [myScrollView setContentSize:CGSizeMake(400,
450)];
}

-(void)textFieldDidEndEditing:(UITextField *)
textField {
    [myScrollView setFrame:CGRectMake(20, 60, 400,
450)];

}

-(BOOL) textFieldShouldReturn:(UITextField *)
textField {
    [textField resignFirstResponder];
    return YES;
}
@end
```

SENDING EMAIL ON iOS

We can send emails using the iOS device's Email App.

Steps required

- Step 1: Make a basic View-based application.

- Step 2: Choose your project file, then targets, and finally MessageUI.framework.
- Step 3: In ViewController.xib, build a button and an action for sending an email.
- Step 4: Make the following changes to ViewController.h:

```
#import <UIKit/UIKit.h>
#import <MessageUI/MessageUI.h>

@interface ViewController : UIViewController<MFMai
lComposeViewControllerDelegate> {
   MFMailComposeViewController *mailComposer;
}

-(IBAction)sendMail:(id)sender;

@end
```

- Step 5: Make the following changes to ViewController.m:

```
#import "ViewController.h"

@interface ViewController ()
@end

@implementation ViewController

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

- (void)didReceiveMemoryWarning {
   [super didReceiveMemoryWarning];
   // Dispose of any resources that can recreate.
}

-(void)sendMail:(id)sender {
   mailComposer = [[MFMailComposeViewController
alloc]init];
   mailComposer.mailComposeDelegate = self;
   [mailComposer setSubject:@"Test mail"];
   [mailComposer setMessageBody:@"Testing message
```

```
    for test mail" isHTML:NO];
    [self presentModalViewController:mailComposer
animated:YES];
}

#pragma mark - mail compose delegate
-(void)mailComposeController:(MFMailComposeViewCon
troller *)controller
    didFinishWithResult:(MFMailComposeResult)result
error:(NSError *)error{

    if (result) {
        NSLog(@"Result : %d",result);
    }

    if (error) {
        NSLog(@"Error : %@",error);
    }

    [self dismissModalViewControllerAnimated:YES];
}
@end
```

We will see the output after hitting Send Email.

AUDIO AND VIDEO IN iOS

Audio and video are pretty popular in today's technologies. It is supported in iOS via the AVFoundation.framework and the MediaPlayer.framework.

Procedures involved

- Step 1: Make a basic View-based application.
- Step 2: Choose your project file, targets, and then AVFoundation. framework and MediaPlayer.framework.
- Step 3: In ViewController.xib, add two buttons and an action for playing audio and video.
- Step 4: Make the following changes to ViewController.h:

```
#import <UIKit/UIKit.h>
#import <AVFoundation/AVFoundation.h>
#import <MediaPlayer/MediaPlayer.h>

@interface ViewController : UIViewController {
```

```
    AVAudioPlayer *audioPlayer;
    MPMoviePlayerViewController *moviePlayer;
}
-(IBAction)playAudio:(id)sender;
-(IBAction)playVideo:(id)sender;
@end
```

- Step 5: Make the following changes to ViewController.m:

```
#import "ViewController.h"

@interface ViewController ()
@end

@implementation ViewController

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

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can recreate.
}

-(IBAction)playAudio:(id)sender {
    NSString *path = [[NSBundle mainBundle]
    pathForResource:@"audioTest" ofType:@"mp3"];
    audioPlayer = [[AVAudioPlayer alloc]
initWithContentsOfURL:
    [NSURL fileURLWithPath:path] error:NULL];
    [audioPlayer play];
}

-(IBAction)playVideo:(id)sender {
    NSString *path = [[NSBundle mainBundle]
pathForResource:
    @"videoTest" ofType:@"mov"];
    moviePlayer = [[MPMoviePlayerViewController
    alloc]initWithContentURL:[NSURL
fileURLWithPath:path]];
    [self presentModalViewController:moviePlayer
animated:NO];
}
@end
```

Please remember that we must include audio and video files to ensure that we receive the desired results.

When we click on the play video button, we will see the output.

We will hear the sounds when we press the play button.

FILE HANDLING IN iOS

Because file management cannot be demonstrated visually using the program, the essential techniques for managing files are detailed here. It is important to note in Objective-C that the application bundle only has read access and that we will not be able to edit the files. You can still make changes to your application's documents directory.

METHODS FOR FILE HANDLING

The techniques for accessing and altering files are detailed further below. We must change the FilePath1, FilePath2, and FilePath strings with our needed complete file paths to achieve the desired action.

Check to See If a File in Objective-C Exists at a Given Path

```
NSFileManager *fileManager = [NSFileManager
defaultManager];

//Get the documents directory
NSArray *directoryPaths =
NSSearchPathForDirectoriesInDomains
(NSDocumentDirectory, NSUserDomainMask, YES);
NSString *documentsDirectoryPath = [directoryPaths
objectAtIndex:0];

if ([fileManager fileExistsAtPath:@""]==YES) {
   NSLog(@"File exists");
}
```

Comparing the Contents of Two Files

```
if ([fileManager contentsEqualAtPath:@"FilePath1"
andPath:@" FilePath2"]) {
   NSLog(@"Same-content");
}
```

Check to See If It Is Writable, Readable, and Executable

```
if ([fileManager isWritableFileAtPath:@"FilePath"]) {
   NSLog(@"isWritable");
}

if ([fileManager isReadableFileAtPath:@"FilePath"]) {
   NSLog(@"isReadable");
}

if ( [fileManager isExecutableFileAtPath:@"FilePath"])
{
   NSLog(@"isExecutable");
}
```

Move File

```
if([fileManager moveItemAtPath:@”FilePath1”
   toPath:@”FilePath2” error:NULL]) {
   NSLog(@”Moved-successfully”);
}
```

Copy File

```
if ([fileManager copyItemAtPath:@"FilePath1"
   toPath:@"FilePath2"  error:NULL]) {
   NSLog(@"Copied-successfully");
}
```

Remove File

```
if ([fileManager removeItemAtPath:@”FilePath”
error:NULL]) {
   NSLog(@”Removed-successfully”);
}
```

Read File

```
NSData *data = [fileManager contentsAtPath:@”Path”];
```

Write File

```
[fileManager createFileAtPath:@"" contents:data
attributes:nil];
```

ACCESSING MAPS ON iOS

Maps are usually helpful for locating places. The MapKit framework is used to incorporate maps into iOS.

Procedures involved

- Step 1: Develop a basic view-based application.
- Step 2: Choose our project file, then targets, and finally MapKit. framework.
- Step 3: We need additionally include the Corelocation.framework.
- Step 4: Insert a MapView into ViewController.xib and create an ibOutlet called mapView.
- Step 5: Select File New File to create a new file. Next, choose the Objective-C class.
- Step 6: Name the class MapAnnotation with the term "subclass of" as NSObject.
- Step 7: Click the Create button.
- Step 8: Make the following changes to MapAnnotation.h:

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

@interface MapAnnotation : NSObject<MKAnnotation>
@property (nonatomic, strong) NSString *title;
@property (nonatomic, readwrite)
CLLocationCoordinate2D coordinate;

- (id)initWithTitle:(NSString *)title
andCoordinate:
   (CLLocationCoordinate2D)coordinate2d;

@end
```

- Step 9: Make the following changes to MapAnnotation.m:

```
#import "MapAnnotation.h"

@implementation MapAnnotation
-(id)initWithTitle:(NSString *)title andCoordinate:
   (CLLocationCoordinate2D)coordinate2d {
```

```
    self.title = title;
    self.coordinate =coordinate2d;
    return self;
}
@end
```

- Step 10: Make the following changes to ViewController.h:

```
#import <UIKit/UIKit.h>
#import <MapKit/MapKit.h>
#import <CoreLocation/CoreLocation.h>

@interface ViewController : UIViewController<MKMap
ViewDelegate> {
    MKMapView *mapView;
}
@end
```

- Step 11: Make the following changes to ViewController.m:

```
#import "ViewController.h"
#import "MapAnnotation.h"

@interface ViewController ()
@end

@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    mapView = [[MKMapView alloc]initWithFrame:
    CGRectMake(20, 110, 320, 310)];
    mapView.delegate = self;
    mapView.centerCoordinate =
CLLocationCoordinate2DMake(38.32, -142.02);
    mapView.mapType = MKMapTypeHybrid;
    CLLocationCoordinate2D location;
    location.latitude = (double) 38.32239;
    location.longitude = (double) -142.023919;

    // Add the annotation to our map view
    MapAnnotation *newAnnotation = [[MapAnnotation
alloc]
    initWithTitle:@"Apple Head quaters"
andCoordinate:location];
```

```
    [mapView addAnnotation:newAnnotation];
    CLLocationCoordinate2D location2;
    location2.latitude = (double) 38.32239;
    location2.longitude = (double) -142.023919;
    MapAnnotation *newAnnotation2 = [[MapAnnotation
alloc]
    initWithTitle:@"Test annotation"
andCoordinate:location2];
    [mapView addAnnotation:newAnnotation2];
    [self.view addSubview:mapView];
}

// When map annotation point is added, zoom it
(1500 range)
- (void)mapView:(MKMapView *)mv
didAddAnnotationViews:(NSArray *)views {
    MKAnnotationView *annotationView = [views
objectAtIndex:0];
    id <MKAnnotation> mp = [annotationView
annotation];
    MKCoordinateRegion region =
MKCoordinateRegionMakeWithDistance
    ([mp coordinate], 1500, 1500);
    [mv setRegion:region animated:YES];
    [mv selectAnnotation:mp animated:YES];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can recreate.
}
@end
```

IN-APP PURCHASE IN iOS

In-App purchases are used to acquire additional content or upgrade an application's features.

The involved steps

- Step 1: Ensure that you have a unique App ID in iTunes Connect and update the application's bundle ID and code signing in Xcode with the corresponding provisioning profile.

- Step 2: Create a new application and update its information. More information is available in Apple's Add new apps documentation.
- Step 3: Add a new product now for In-App purchase to your application's Manage In-App Purchase page.
- Step 4: Ensure that the bank information for your application is set up. This must be configured for In-App purchase to function. Create a test user account using the Manage Users option on the App's iTunes Connect page.
- Step 5: The following steps involve handling code and designing the user interface for our In-App purchase.
- Step 6: Create a single-view application with the bundle identifier specified in iTunes connect.
- Step 7: Update the ViewController.xib file.
- Step 8: Create IBOutlets for the three labels and the button. Name them productTitleLabel, productDescriptionLabel, productPriceLabel, and purchaseButton.
- Step 9: Choose our project file, then targets, and finally StoreKit. framework.
- Step 10: Make the following changes to ViewController.h:

```
#import <UIKit/UIKit.h>
#import <StoreKit/StoreKit.h>

@interface ViewController : UIViewController<
SKProductsRequestDelegate,SKPaymentTransactionObse
rver> {
   SKProductsRequest *productsRequest;
   NSArray *validProducts;
   UIActivityIndicatorView *activityIndicatorView;
   IBOutlet UILabel *productTitleLabel;
   IBOutlet UILabel *productDescriptionLabel;
   IBOutlet UILabel *productPriceLabel;
   IBOutlet UIButton *purchaseButton;
}

- (void)fetchAvailableProducts;
- (BOOL)canMakePurchases;
```

```
- (void)purchaseMyProduct:(SKProduct*)product;
- (IBAction)purchase:(id)sender;

@end
```

- Step 11: Make the following changes to ViewController.m:

```
#import "ViewController.h"
#define kTutorialPointProductID
@"com.tutorialPoints.testApp.testProduct"

@interface ViewController ()
@end

@implementation ViewController

- (void)viewDidLoad {
   [super viewDidLoad];

   // Adding the activity indicator
   activityIndicatorView =
[[UIActivityIndicatorView alloc]
   initWithActivityIndicatorStyle:UIActivity
IndicatorViewStyleWhiteLarge];
   activityIndicatorView.center = self.view.
center;
   [activityIndicatorView hidesWhenStopped];
   [self.view addSubview:activityIndicatorView];
   [activityIndicatorView startAnimating];

   //Hide the purchase button initially
   purchaseButton.hidden = YES;
   [self fetchAvailableProducts];
}

- (void)didReceiveMemoryWarning {
   [super didReceiveMemoryWarning];
   // Dispose of any resources that can recreate.
}

-(void)fetchAvailableProducts {
   NSSet *productIdentifiers = [NSSet
   setWithObjects:kTutorialPointProductID,nil];
```

```
    productsRequest = [[SKProductsRequest alloc]
    initWithProductIdentifiers:productIdentifiers];
    productsRequest.delegate = self;
    [productsRequest start];
}

- (BOOL)canMakePurchases {
    return [SKPaymentQueue canMakePayments];
}

- (void)purchaseMyProduct:(SKProduct*)product {
    if ([self canMakePurchases]) {
        SKPayment *payment = [SKPayment
paymentWithProduct:product];
        [[SKPaymentQueue defaultQueue]
addTransactionObserver:self];
        [[SKPaymentQueue defaultQueue]
addPayment:payment];
    } else {
        UIAlertView *alertView = [[UIAlertView
alloc]initWithTitle:
        @"Purchases are disabled in your device"
message:nil delegate:
        self cancelButtonTitle:@"Ok"
otherButtonTitles: nil];
        [alertView show];
    }
}
-(IBAction)purchase:(id)sender {
    [self purchaseMyProduct:[validProducts
objectAtIndex:0]];
    purchaseButton.enabled = NO;
}

#pragma mark StoreKit Delegate

-(void)paymentQueue:(SKPaymentQueue *)queue
updatedTransactions:(NSArray *)transactions {
    for (SKPaymentTransaction *transaction in
transactions) {
        switch (transaction.transactionState) {
            case SKPaymentTransactionStatePurchasing:
```

```
            NSLog(@"Purchasing");
         break;

         case SKPaymentTransactionStatePurchased:
            if ([transaction.payment.
productIdentifier
            isEqualToString:kTutorialPointProduc
tID]) {
               NSLog(@"Purchased ");
               UIAlertView *alertView =
[[UIAlertView alloc]initWithTitle:
               @"Purchase is completed
succesfully" message:nil delegate:
               self cancelButtonTitle:@"Ok"
otherButtonTitles: nil];
               [alertView show];
            }
            [[SKPaymentQueue defaultQueue] finishT
ransaction:transaction];
         break;

         case SKPaymentTransactionStateRestored:
            NSLog(@"Restored ");
            [[SKPaymentQueue defaultQueue] finishT
ransaction:transaction];
         break;

         case SKPaymentTransactionStateFailed:
            NSLog(@"Purchase failed ");
         break
         default:
         break;
      }
   }
}

-(void)productsRequest:(SKProductsRequest *)
request
didReceiveResponse:(SKProductsResponse *)response
{
   SKProduct *validProduct = nil;
   int counts = [response.products count];
```

```
   if (counts>0) {
      validProducts = response.products;
      validProducts = [response.products
objectAtIndex:0];

      if ([validProduct.productIdentifier
         isEqualToString:kTutorialPointProductID])
{
         [productTitleLabel setText:[NSString
stringWithFormat:
            @"Product Title: %@",validProduct.
localizedTitle]];
         [productDescriptionLabel
setText:[NSString stringWithFormat:
            @"Product Desc: %@",validProduct.
localizedDescription]];
         [productPriceLabel setText:[NSString
stringWithFormat:
            @"Product Price: %@",validProduct.
price]];
      }
   } else {
      UIAlertView *tmp = [[UIAlertView alloc]
         initWithTitle:@"Not-Available"
         message:@"No products to purchased"
         delegate:self
         cancelButtonTitle:nil
         otherButtonTitles:@"Ok", nil];
         [tmp show];
   }

   [activityIndicatorView stopAnimating];
   purchaseButton.hidden = NO;
}
@end
```

Note: We must change the value of kTutorialPointProductID to the productID we established for our In-App Purchase. We may add several products by modifying the NSSet of productIdentifiers in fetchAvailable-Products. Handle the purchase-related actions for the product IDs we add in the same way.

Check that we have logged out of your account from the settings screen. Select Use Existing Apple ID when we click the Initiate Purchase button. Enter the correct username and password for our test account. In a few seconds, we will get the following notice.

We will receive an alert once our product has been successfully purchased. Where we show this notice, we may view the necessary code for changing the application functionality.

iAd INTEGRATION IN iOS

The apple server serves advertisements using iAd. iAd assists us in generating income from an iOS app.

Steps involved in iAd integration

- Step 1: Develop a basic view-based application.
- Step 2: Select our project file, then targets, and finally iAd.framework in frameworks.
- Step 3: Make the following changes to ViewController.h:

```
#import <UIKit/UIKit.h>
#import <iAd/iAd.h>

@interface ViewController : UIViewController<ADBan
nerViewDelegate> {
   ADBannerView *bannerView;
}
@end
```

- Step 4: Make the following changes to ViewController.m:

```
#import "ViewController.h"

@interface ViewController ()
@end

@implementation ViewController

- (void)viewDidLoad {
   [super viewDidLoad];
   bannerView = [[ADBannerView alloc]
initWithFrame:
   CGRectMake(0, 0, 330, 60)];
```

```
    // Optional to set background color to clear
the color
    [bannerView setBackgroundColor:[UIColor
clearColor]];
    [self.view addSubview: bannerView];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can recreate.
}

#pragma mark - AdViewDelegates

-(void)bannerView:(ADBannerView *)banner
    didFailToReceiveAdWithError:(NSError *)error {
    NSLog(@"The Error loading");
}

-(void)bannerViewDidLoadAd:(ADBannerView *)banner
{
    NSLog(@"The Ad loaded");
}

-(void)bannerViewWillLoadAd:(ADBannerView *)banner
{
    NSLog(@"The Ad will load");
}

-(void)bannerViewActionDidFinish:(ADBannerView *)
banner {
    NSLog(@"The Ad did finish");
}
@end
```

GameKit IN iOS

Gamekit is a framework that gives a leaderboard, achievements, and more features to an iOS application. In this tutorial, we will describe the procedures required to install a leaderboard and change the score.

The involved steps

- Step 1: Ensure that we have a unique App ID in iTunes Connect and update the application's bundle ID and code signing in Xcode with the relevant provisioning profile.

- Step 2: Create a new application and edit its details. We may know more about this in apple-add new applications documentation.
- Step 3: Set up a leaderboard in the Manage Game Center of our application's page where add a single leaderboard and specify leaderboard ID and score Type. Here we supply leader board ID.
- Step 4: The following stages are connected to handling code and developing UI for our application.
- Step 5: Create a single view application and input the bundle identification as the identifier given in iTunes connect.
- Step 6: Update the ViewController.xib.
- Step 7: Choose our project file, then targets, and finally GameKit. framework.
- Step 8: Create IBActions for the newly inserted buttons.
- Step 9: Make the following changes to the ViewController.h file:

```
#import <UIKit/UIKit.h>
#import <GameKit/GameKit.h>

@interface ViewController : UIViewController
<GKLeaderboardViewControllerDelegate>

-(IBAction)updateScore:(id)sender;
-(IBAction)showLeaderBoard:(id)sender;

@end
```

- Step 10: Make the following changes to ViewController.m:

```
#import "ViewController.h"

@interface ViewController ()
@end

@implementation ViewController

- (void)viewDidLoad {
   [super viewDidLoad];
   if([GKLocalPlayer localPlayer].authenticated ==
NO) {
```

```
      [[GKLocalPlayer localPlayer]
      authenticateWithCompletionHandler:^(NSError
*error) {
         NSLog(@"Error%@",error);
      }];
   }
}

- (void)didReceiveMemoryWarning {
   [super didReceiveMemoryWarning];
   // Dispose of any resources that can recreate.
}

- (void) updateScore: (int64_t) score
   forLeaderboardID: (NSString*) category {
   GKScore *scoreObj = [[GKScore alloc]
   initWithCategory:category];
   scoreObj.value = score;
   scoreObj.context = 0;

   [scoreObj reportScoreWithCompletionHandler:^(NS
Error *error) {
      // Completion code can be added here
      UIAlertView *alert = [[UIAlertView alloc]
      initWithTitle:nil message:@"Score Updated
Succesfully"
      delegate:self cancelButtonTitle:@"Ok"
otherButtonTitles: nil];
      [alert show];
   }];
}

-(IBAction)updateScore:(id)sender {
   [self updateScore:200 forLeaderboardID:@"tutori
alsPoint"];
}

-(IBAction)showLeaderBoard:(id)sender {
   GKLeaderboardViewController
*leaderboardViewController =
   [[GKLeaderboardViewController alloc] init];
   leaderboardViewController.leaderboardDelegate =
self;
```

```
    [self presentModalViewController:
    leaderboardViewController animated:YES];
}

#pragma mark - Gamekit delegates
- (void)leaderboardViewControllerDidFinish:
(GKLeaderboardViewController *)viewController {
    [self dismissModalViewControllerAnimated:YES];
}
@end
```

STORYBOARDS IN iOS

iOS 5 introduces storyboards. When we employ storyboards, we should aim for a deployment goal of 5.0 or above. Storyboards assist us in creating all of the screens of an application and connecting them under a single interface MainStoryboard.storyboard. It also aids in the reduction of coding for pushing and showing view controllers.

Procedures involved

- Step 1: Create a single-view application and ensure that the storyboard checkbox is selected while creating the application.
- Step 2: Choose MainStoryboard.storyboard with a single view controller. Update the view controllers after adding one more.
- Step 3: Now, link the two view controllers. Right-click the "show modal" button and move it to the left-side view controller's right-view controller.
- Step 4: Choose a modal from the three options displayed.
- Step 5: Make the following changes to ViewController.h:

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

@interface ViewController : UIViewController

-(IBAction)done:(UIStoryboardSegue *)seque;

@end
```

- Step 6: Make the following changes to ViewController.m:

```
#import "ViewController.h"
```

```
@interface ViewController ()
@end

@implementation ViewController

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

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can recreate.
}

-(IBAction)done:(UIStoryboardSegue *)seque {
    [self.navigationController popViewControllerAni
mated:YES];
}

@end
```

- Step 7: Choose the MainStoryboard.storyboard and right-click on the Exit button on the right side view controller, then pick done and link with the back button.

AUTO LAYOUTS IN iOS

In iOS 6.0, auto-layouts were added. We should have a deployment goal of 6.0 or greater when using auto-layouts. Auto-layouts allow us to design interfaces compatible with many orientations and devices.

Aim of Our Example

We will add two buttons positioned at a specific distance from the screen's center. Additionally, we will attempt to include a resizable text box positioned a specific distance above the buttons.

Our Strategy

We will add a text box, two buttons, and their respective restrictions to the code. Each UI Element's restrictions will be generated and applied to the super view. To get the desired outcome, we must deactivate auto-resizing for each UI item we add.

The Involved Steps

- Step 1: Create a basic view-based application.
- Step 2: We will just modify ViewController.m, as seen below:

```
#import "ViewController.h"

@interface ViewController ()
@property (nonatomic, strong) UIButton *leftButton;
@property (nonatomic, strong) UIButton *rightButton;
@property (nonatomic, strong) UITextField *textfield;

@end
@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    UIView *superview = self.view;

    /*1. Create leftButton and add to view*/
    self.leftButton = [UIButton buttonWithType:UIButton
TypeRoundedRect];
    self.leftButton.
translatesAutoresizingMaskIntoConstraints = NO;
    [self.leftButton setTitle:@"LeftButton" forState:UI
ControlStateNormal];
    [self.view addSubview:self.leftButton];

    /* 2. Constraint to the position LeftButton's X*/
    NSLayoutConstraint *leftButtonXConstraint =
[NSLayoutConstraint
    constraintWithItem:self.leftButton attribute:NSLayo
utAttributeCenterX
    relatedBy:NSLayoutRelationGreaterThanOrEqual
toItem:superview attribute:
    NSLayoutAttributeCenterX multiplier:1.0
constant:-60.0f];

    /* 3. Constraint to the position LeftButton's Y*/
    NSLayoutConstraint *leftButtonYConstraint =
[NSLayoutConstraint
```

```
   constraintWithItem:self.leftButton attribute:NSLayo
utAttributeCenterY
   relatedBy:NSLayoutRelationEqual toItem:superview
attribute:
   NSLayoutAttributeCenterY multiplier:1.0f
constant:0.0f];

   /* 4. Add constraints to the button's superview*/
   [superview addConstraints:@[ leftButtonXConstraint,
   leftButtonYConstraint]];

   /*5. Create the rightButton and add to our view*/
   self.rightButton = [UIButton buttonWithType:UIButto
nTypeRoundedRect];
   self.rightButton.
translatesAutoresizingMaskIntoConstraints = NO;
   [self.rightButton setTitle:@"RightButton" forState:
UIControlStateNormal];
   [self.view addSubview:self.rightButton];

   /*6. Constraint to the position RightButton's X*/
   NSLayoutConstraint *rightButtonXConstraint =
[NSLayoutConstraint
   constraintWithItem:self.rightButton attribute:NSLay
outAttributeCenterX
   relatedBy:NSLayoutRelationGreaterThanOrEqual
toItem:superview attribute:
   NSLayoutAttributeCenterX multiplier:1.0
constant:60.0f];

   /*7. Constraint to the position RightButton's Y*/
   rightButtonXConstraint.priority =
UILayoutPriorityDefaultHigh;
   NSLayoutConstraint *centerYMyConstraint =
[NSLayoutConstraint
   constraintWithItem:self.rightButton attribute:NSLay
outAttributeCenterY
   relatedBy:NSLayoutRelationGreaterThanOrEqual
toItem:superview attribute:
   NSLayoutAttributeCenterY multiplier:1.0f
constant:0.0f];
   [superview addConstraints:@[centerYMyConstraint,
   rightButtonXConstraint]];
```

```
   //8. Add the Text field
   self.textfield = [[UITextField alloc]initWithFrame:
   CGRectMake(0, 100, 100, 30)];
   self.textfield.borderStyle =
UITextBorderStyleRoundedRect;
   self.textfield.
translatesAutoresizingMaskIntoConstraints = NO;
   [self.view addSubview:self.textfield];

   //9. The Text field Constraints
   NSLayoutConstraint *textFieldTopConstraint =
[NSLayoutConstraint
   constraintWithItem:self.textfield
attribute:NSLayoutAttributeTop
   relatedBy:NSLayoutRelationGreaterThanOrEqual
toItem:superview
   attribute:NSLayoutAttributeTop multiplier:1.0
constant:60.0f];
   NSLayoutConstraint *textFieldBottomConstraint =
[NSLayoutConstraint
   constraintWithItem:self.textfield
attribute:NSLayoutAttributeTop
   relatedBy:NSLayoutRelationGreaterThanOrEqual
toItem:self.rightButton
   attribute:NSLayoutAttributeTop multiplier:0.8
constant:-60.0f];
   NSLayoutConstraint *textFieldLeftConstraint =
[NSLayoutConstraint
   constraintWithItem:self.textfield
attribute:NSLayoutAttributeLeft
   relatedBy:NSLayoutRelationEqual toItem:superview
attribute:
   NSLayoutAttributeLeft multiplier:1.0
constant:30.0f];
   NSLayoutConstraint *textFieldRightConstraint =
[NSLayoutConstraint
   constraintWithItem:self.textfield
attribute:NSLayoutAttributeRight
   relatedBy:NSLayoutRelationEqual toItem:superview
attribute:
   NSLayoutAttributeRight multiplier:1.0
constant:-30.0f];
```

```
    [superview addConstraints:@
[textFieldBottomConstraint,
    textFieldLeftConstraint, textFieldRightConstraint,
    textFieldTopConstraint]];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can recreate.
}
@end
```

Important notes: In steps 1, 5, and 8, we simply created two buttons and a text field programmatically.

In the following stages, we set constraints and applied them to the relevant super views, essentially self-views. The restrictions of one of the left buttons are depicted here.

```
NSLayoutConstraint *leftButtonXConstraint =
[NSLayoutConstraint
constraintWithItem:self.leftButton attribute:NSLayoutA
ttributeCenterX
relatedBy:NSLayoutRelationGreaterThanOrEqual
toItem:superview attribute:
NSLayoutAttributeCenterX multiplier:1.0
constant:-70.0f];
```

We have constraintWithItem and toItem, which determine the UI components we want to limit. The attribute determines how the two items are connected. "relatedBy" determines how much influence the characteristics have on the components. The multiplier is the multiplication factor, and the constant is added.

In the above example, the X of leftButton is always bigger than or equal to −60 pixels concerning the super view's center. Other limitations are defined in the same way.

TWITTER AND FACEBOOK ON iOS

Twitter is now included in iOS 5.0, while Facebook is integrated into iOS 6.0. The classes given by Apple are used in our tutorial, and the deployment targets for Twitter and Facebook are iOS 5.0 and iOS 6.0, respectively.

Steps required

- Step 1: Develop a basic view-based application.
- Step 2: Select your project file, then targets, and add Social.framework and Accounts.framework to the frameworks list.
- Step 3: Create ibActions for two buttons named facebookPost and twitterPost.
- Step 4: Make the following changes to ViewController.h:

```
#import <Social/Social.h>
#import <Accounts/Accounts.h>
#import <UIKit/UIKit.h>

@interface ViewController : UIViewController

-(IBAction)twitterPost:(id)sender;
-(IBAction)facebookPost:(id)sender;

@end
```

- Step 5: Make the following changes to ViewController.m:

```
#import "ViewController.h"

@interface ViewController ()
@end

@implementation ViewController

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

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can recreate.
}

-(IBAction)facebookPost:(id)sender {
    SLComposeViewController *controller =
[SLComposeViewController
```

```
   composeViewControllerForServiceType:SLServiceTy
peFacebook];
   SLComposeViewControllerCompletionHandler
myBlock =
      ^(SLComposeViewControllerResult result){

      if (result ==
SLComposeViewControllerResultCancelled) {
         NSLog(@"Cancelled");
      } else {
         NSLog(@"Done");
      }
      [controller dismissViewControllerAnimated:
YES completion:nil];
   };
   controller.completionHandler = myBlock;

   //Adding Text to the facebook post value from
iOS
   [controller setInitialText:@"My test post"];

   //Adding the URL to the facebook post value
from the iOS
   [controller addURL:[NSURL
URLWithString:@"http://www.test.com"]];

   //Adding Text to the facebook post value from
the iOS
   [self presentViewController:controller
animated:YES completion:nil];
}

-(IBAction)twitterPost:(id)sender {
   SLComposeViewController *tweetSheet =
[SLComposeViewController
   composeViewControllerForServiceType:SLServiceTy
peTwitter];
   [tweetSheet setInitialText:@"My test tweet"];
   [self presentModalViewController:tweetSheet
animated:YES];
}
@end
```

MEMORY MANAGEMENT IN iOS

Initially, iOS's memory management was non-ARC (Automatic Reference Counting), requiring us to retain and release objects. Now that it supports ARC, we no longer need to retain and release objects. Compilation time is handled automatically by Xcode.

MEMORY MANAGEMENT CHALLENGES

According to Apple documentation, the two most significant memory management issues are:

- Releasing or erasing data that is currently in use. It typically results in an application crash or corrupted user data.
- Memory leaks are caused by not releasing inactive data. Memory leaks occur when allocated memory is not released even though it is not used again. Leaks cause your application to consume ever-increasing amounts of memory, which may lead to poor system performance or (in iOS) the termination of our application.

RULES FOR MEMORY MANAGEMENT

- We are responsible for releasing the items we generate when they are no longer required.
- Retain is used to acquire ownership of an object you did not create. You must also release these things when they are no longer required.
- Do not release the items we do not possess.

DEALING WITH MEMORY IN ARC

In ARC, we do not need to employ release and retain. Therefore, all of its associated objects will be freed when the view controller is deleted. Similarly, sub-objects of any object are released when their parent object is released. Note that the whole class will not be released if other classes hold a strong reference to an object of a class. Therefore, poor qualities are advised for delegates.

MEMORY MANAGEMENT TOOLS

With the use of Xcode instrumentation, memory utilization is analyzed. It has features like Activity Monitor, Allocations, Leaks, and Zombies.

ANALYTICAL METHODS FOR MEMORY ALLOCATIONS

- Step 1: Launch an existing application.
- Step 2: Select Product and then Profile.
- Step 3: On the following page, pick Allocations and then Profile.
- Step 4: We'll look at memory allocation for various objects.
- Step 5: We can switch between view controllers to see if the memory is appropriately released.
- Step 6: Similarly, we may utilize Activity Monitor instead of Allocations to observe the total Memory assigned to the application.
- Step 7: We can access our memory use and pinpoint any leaks using these tools.

APPLICATION DEBUGGING IN iOS

While developing an application, we may make mistakes that lead to many faults. We must debug the program to correct these issues or flaws.

CHOOSING A DEBUGGER

Xcode includes two debuggers, notably GDB and LLDB. GDB is the default database type. LLDB is a debugger included in the open-source LLVM compiler project. The "edit active schemes" option allows you to modify the debugger.

HOW TO LOCATE CODING ERRORS

We must construct your application and compile the code to identify coding-related issues. If the code includes errors, the compiler will show all messages, errors, warnings, and potential causes.

Click Product followed by Analyze to spot possible flaws in an application.

SET BREAKPOINTS

Breakpoints enable us to determine the various states of our application's objects, facilitating the identification of several problems, including logical errors. Simply clicking on the line number will establish a breakpoint. To remove a breakpoint, click and drag it out of the document.

When we launch the program and press the playVideo button, it will halt at the line number where we put the breakpoint. It allows users to assess the current condition of the application when the breakpoint is hit.

It is simple to determine which thread triggered the breakpoint. You can see objects like self, sender, and so on down the bottom, which carry the values of the relevant objects, and we can expand some of these objects to see their state.

Click the continue button (leftmost button) to proceed with the application.

When we launch the program and press the playVideo button, it will halt at the line number where we put the breakpoint. It allows us to assess the current state of the application when the breakpoint is hit.

It is simple to determine which thread triggered the breakpoint. You can see objects like self, sender, and so on down the bottom, which carry the values of the relevant objects, and we can expand some of these objects to see their state.

Click the continue button (leftmost button) to proceed with the application.

BREAKPOINT EXCEPTION

We also have exception breakpoints, which cause an application to halt where the exception occurs. After selecting the debug navigator, we can put exception breakpoints by clicking the Add button. Then we must choose Add Exception Breakpoint.

IN AN iOS App, WE MAY USE GOOGLE APIs

Some Google services in Objective-C, such as Drive, Gmail, and many more, have public APIs used to build apps that enable users to interact with their data in these services. To use these services, applications must implement one of the OAuth 2.0 client flows to gain user approval and access tokens that provide access to the APIs.[1]

To obtain access tokens for the signed-in user, we may use the Google Sign-In library, which implements the OAuth 2.0 procedure.

Before we begin

We must first finish the basic Google Sign-In integration.

1. **Determine which scopes have been granted:** Before calling a Google API, use the grantedScopes property of GIDGoogleUser to see which scopes have been given to our app.

```
NSString *driveScope = @"https://www.googleapis.
com/auth/drive.readonly";

// Check if user has granted Drive scope
if (![user.grantedScopes
containsObject:driveScope]) {
  // request the additional drive scope
}
```

Whether or not the user has authorized a specific scope, we may need to request an additional scope to support a specific interaction.

2. **Ask for extra scopes:** If we need to request more scopes, use the addScopes:presentingViewController:callback or addScopes:presentingWindow:callback methods to ask the user to allow our app to be more accessible.

   To request read-only access to a user's Drive via Objective-C, for example:

```
NSArray *additionalScopes = @[ @"https://www.
googleapis.com/auth/drive.readonly" ];
[GIDSignIn.sharedInstance
addScopes:additionalScopes
           presentingViewController:self

callback:^(GIDGoogleUser * _Nullable user,
                                      NSError *
_Nullable error) {
    if (error) { return; }
    if (user == nil) { return; }

    // Check if the user granted access to the
scopes we requested.
}];
```

3. **Make an API call using new tokens:** Wrap your Google API requests in a doWithFreshTokens: block to guarantee that they always have valid access tokens.

```
[user.authentication doWithFreshTokens:^(GIDAuthen
tication * _Nullable authentication,
                                         NSError *
_Nullable error) {
```

```
    if (error) { return; }
    if (authentication == nil) { return; }

    // Get an access token to attach it to a REST
or gRPC request.
    NSString *accessToken = authentication.
accessToken;

    // Or, get object that conforms to
GTMFetcherAuthorizationProtocol for
    // use with GTMAppAuth and Google APIs client
library.
    id<GTMFetcherAuthorizationProtocol> authorizer
= [authentication fetcherAuthorizer];
}];
```

Call the API with the access token by including it in the header of a REST or gRPC request (Authorization: Bearer ACCESS TOKEN) or by using the fetcher authorizer with the Google APIs Client Library.

This chapter covered iOS in Objective-C and used Google APIs with its relevant code.

NOTE

1. Google Identity

BIBLIOGRAPHY

1. How to use Object C in IOS? – https://intellipaat.com/blog/tutorial/ios-tutorial/objective-c/#:~:text=Objective%20C%20is%20used%20in,functions%2C%20and%20control%20flow%20constructs, accessed on May 10, 2022.
2. iOS - Objective-C – https://www.tutorialspoint.com/ios/ios_objective_c.htm, accessed on May 10, 2022.
3. Building your first iOS application – https://livebook.manning.com/book/objective-c-fundamentals/chapter-1/, accessed on May 10, 2022.
4. iOS – Objective-C -https://developer.mixpanel.com/docs/ios-objective-c-quickstart, accessed on May 10, 2022.
5. How do I create delegates in Objective-C? – https://stackoverflow.com/questions/626898/how-do-i-create-delegates-in-objective-c#:~:text=To%20create%20one%2C%20you%20define,as%20implementing%20the%20delegate%20protocol.&text=Then%20you%20could%20create%20an,MyClass%20alloc%5D%20init%5D%3B%20myWebView, accessed on May 10, 2022.
6. Objective-C Delegation by Example – https://eezytutorials.com/ios/objective-c/objective-c-delegation-by-example.php#.Y1RG63ZBzIU, accessed on May 10, 2022.

7. iOS – UI Elements - https://www.tutorialspoint.com/ios/ios_ui_elements.htm, accessed on May 11, 2022.
8. Introducing Objective-C UI – https://www.swiftbysundell.com/special/introducing-objective-c-ui/, accessed on May 11, 2022.
9. Basic UI Elements – https://www.coursera.org/lecture/ios-app-development-basics/basic-ui-elements-zgBda, accessed on May 11, 2022.
10. iOS – Accelerometer – https://www.tutorialspoint.com/ios/ios_accelerometer.htm, accessed on May 11, 2022.
11. Sending Email using the iOS application – https://www.javatpoint.com/sending-email-using-ios-application, accessed on May 11, 2022.
12. iOS – Accessing Maps – https://www.tutorialspoint.com/ios/ios_accessing_maps.htm, accessed on May 12, 2022.
13. iOS – iAd Integration – https://www.tutorialspoint.com/ios/ios_iad_integration.htm, accessed on May 12, 2022.
14. Auto Layout Basics – https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwjdqPj8y_T6AhUIRmwGHQlDDwYQFnoECCMQAQ&url=https%3A%2F%2Fguides.codepath.com%2Fios%2FAuto-Layout-Basics&usg=AOvVaw2KhY1Yb7RSSsvTGEDRTNB4, accessed on May 12, 2022.
15. Constraints programmatically with Objective-C – https://stackoverflow.com/questions/51989666/constraints-programmatically-with-objective-c, accessed on May 12, 2022.
16. Building an App with Only Code Using Auto Layout – https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwjhv-CIzPT6AhX6TWwGHQBAAScQFnoECCAQAQ&url=https%3A%2F%2Fwww.raywenderlich.com%2F6004856-building-an-app-with-only-code-using-auto-layout&usg=AOvVaw3nIluzeAAJD2dDvpbBefAe, accessed on May 12, 2022.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

CHAPTER 4

Functional Programming

IN THIS CHAPTER

- Writing Methods
- Project-based Samples

In the previous chapter, we discussed interface and API in Objective-C, and in this chapter, we will cover functional programming.

Object-functional programming is a programming technique that emphasizes object transformation and employs both object-oriented and functional tools and ideas to do this. The creation of effects in general and the modification of objects and object references in particular rarely occurs, as transparently as feasible to other parts of the code and as far away from transformational code as practicable.

Data and behavior are brought together, and access modifiers are utilized to enable only reasonable dependencies wherever feasible. Functional features and many features traditionally provided more frequently by functional languages are used to avoid mutation of object and object references as much as possible and easily remain modular even in the small, for example, with functions as a lightweight alternative to the Strategy Pattern. As is typical for functional programming, the software is separated into code mostly devoid of effects and code-producing effects, with effects being as transparent to other parts of the code as possible; this ensures that large portions of the code are unaffected by the disadvantages that effects entail, such as for local understandability and modular composability.

DOI: 10.1201/9781003357506-4

In contrast to object orientation, objects are often immutable, at least from the outside; methods typically calculate but do not cause visible consequences. In contrast to functional programming, functions operate less on passive data structures and more frequently as methods belonging to objects or object types with access to their private data and behavior.

Appropriate mechanisms are utilized to achieve key outcomes, such as displaying data on the screen; these mechanisms might take on various forms.

WHY OBJECT-FUNCTIONAL PROGRAMMING?

Programming is not about implementing desired behavior in an "arbitrary" manner. The creation and improvement of the software should be efficient, and the outcomes should be of good quality. Object-oriented and functional programming assist this objective in their unique ways.

On the one hand, this approach develops positive characteristics, such as local understandability and modular composability. On the other hand, its typically imperative emphasis hinders these positive characteristics. In addition, mutable objects are not easily employed safely in a parallel setting, and pure object orientation is sometimes too cumbersome for tiny applications. In contrast, the functional method is distinguished by its lightweight characteristics and its restriction and isolation of effects.

The functional method provides fewer tools for organizing software systems, abstracting data, and eliminating unwanted dependencies. In contrast, the object-oriented method provides effective replies with its associated objects and properties.

Therefore, the advantages of one strategy are the disadvantages of the other. Combining both strategies combines their strengths and compensates for their respective deficiencies. We get the potential benefits of both without their downsides. The amount to which this promise is fulfilled is also contingent on the language used.

OBJECTIVE-C FUNCTIONAL PROGRAMMING

This is called Objective-C:

```
@interface NSArray (Map)
DefineMethod(map, id (^)(id), NSArray*);
DefineMethod(reduce, id (^(^)(id))(id), NSArray*);
@end
```

```
@interface NSString (Funcs)
DefineProperty(uppercaseString, NSString*);
DefineMethod(plus, NSString*, NSString*);
@end

int main()
{
  Print(@[@"foo", @"bar", @"baz"].map(NSString.
uppercaseString));
  // -> ( FOO, BAR, BAZ )

  Print(@"Hello".plus(@" I Miss you"));
  // -> Hello I Miss you

  Print(@[@"foo", @"bar", @"baz"].reduce(NSString.
plus));
  // -> foobarbaz

  InstanceProperty massUppercaser = NSArray.
map(NSString.uppercaseString);
  Print(massUppercaser(@[@"abc",@"def"]));
  // -> ( ABC, DEF )

  InstanceProperty exclamate = NSString.plus(@"!!");
  Print(@[@"quick", @" to batmobile"].map(NSString.
uppercaseString).map(exclamate).reduce(NSString.
plus));
  // -> QUICK! TO THE BATMOBILE!
}
```

It does compile and run.

The methods NSArray.map, NSArray.reduce, and NSString.plus are implemented as standard Objective-C methods:

```
@implementation NSString (Plus)
- (NSString*) plus:(NSString*) string
{
    return [self stringByAppendingString:string];
}
@end
```

The basic concept is to build different, parameter-less methods (such as -[NSString plus] instead of -[NSString plus:]) that return a block that

accepts an argument used as the parameter of the real method. It's OK if we need to reread that phrase, but it's not that difficult.

Building block wrapper implementations for every single method would be tedious. With resolveInstanceMethod, the objective-C runtime allows us to add methods on demand. This was almost shockingly uncomplicated.

The whole point is made here. This hack is likely filled with problems, so do not use it for anything important.

A few random thoughts we picked up along the way:

On Functional Programming

exclamate = NSString.plus(@"!!"); is a partial application of the NSString.plus function in Functional Programming. It generates a new function with one less parameter. (Does it refer to currying or partial application? We're not sure; my impression is that it's the same for functions with two or fewer parameters.)

We're a noob at functional programming. These block attributes are most likely referred to as monoids.

On the ObjC Runtime

To add class methods at runtime in Objective-C, adding instance methods to the metaclass is necessary. Class methods are the instance methods of a class's metaclass.

Similarly, resolveClassMethod might be disregarded for the most part. When resolveClassMethod fails, the runtime runs the metaclass's resolveInstanceMethod.

Because I care about my mental health, we only included support for object (id) argument and return types. Using NSInvocation and va args, we began experimenting with methods with more than one parameter, but things were getting out of hand.

Because YOLO, we're swizzling [NSObject resolveInstanceMethod]. This is strong; it is comparable to modifying the dispatch mechanism of the language.

Clang supports the dot notation for attributes and method signatures with no arguments, including Class methods. Class.method may replace [Class method]. However, they do not support autocomplete.

Because we care about our personal wellbeing, we only included support for object (id) argument and return types. Using NSInvocation and va args, we had begun playing with methods with>1 parameter, but things were getting out of hand.

On Objective-C and Language Design

I've said before that Objective-C is quite adaptable. Overall, it does not complain excessively: dynamic languages rule. As usual, the least entertaining aspect is the stupid block syntax.

It is enjoyable to use a programming language with an entirely different coding style than what it was created for. However, if you examine Objective-C source files from the mid-1990s and compare them to modern UIKit, you will see that they are not the same language. In old-style ObjC, the id type was inferred for arguments and return types (it still functions), and methods often returned self rather than void.[1]

In the Objective-C community, Higher-Order Messaging is hardly a novel notion. Of course, there are the KVC collection operators, such as [items valueForKeyPath:@"@sum.price"]; the mere existence of NSInvocation demonstrates that language maintainers have long been interested in this area.

Before UIKit and stackoverflow, cocoadev.com had this page: http://cocoadev.com/HigherOrderMessaging.

We wonder what the Cocoa Target-Action paradigm might look like if it weredesigned with Functional Programming in mind. The last consideration is that we implemented block methods as read-only attributes. What if method properties were read/write? At runtime, we might write NSString.plus = <something else> for the whole class or for a particular instance.

WRITE OBJECTIVE-C CODE

If we have never developed for iOS or Mac OS X, we must familiarize ourselves with Objective-C, the major programming language. Objective-C is not a difficult programming language, and once we get familiar with it, we will recognize its beauty. The Objective-C programming language permits complex object-oriented programming. It expands the conventional ANSI C programming language by adding syntax for class and method definitions. Additionally, it encourages the dynamic expansion of classes and interfaces, which any class may adopt.

If we are acquainted with ANSI C, the material below should assist you in learning the fundamental syntax of Objective-C. And if we've coded in other object-oriented languages, we'll discover that Objective-C supports

many conventional object-oriented principles, like encapsulation, inheritance, and polymorphism.

The Objective-C Programming Language provides a comprehensive explanation of the Objective-C language.

Objective-C language is the superset of the C programming language.

The Objective-C programming language defines a syntax for establishing classes and methods, invoking object methods, dynamically extending classes, and developing problem-specific programming interfaces. Being a superset of the C programming language, Objective-C offers the same syntax as C. We have access to all the standard components, including basic types (int, float, etc.), structures, functions, pointers, and control-flow constructs like if…else and for expressions. In addition, we have access to the standard C library functions, such as those defined in stdlib.h and stdio.h.

Objective-C extends ANSI C with the following syntax and features:

- Introduction of new classes
- Class and instance methods
- Calling a method (called messaging)
- Declaration of attributes (and automatic synthesizing of accessor methods from them)
- Static and dynamic typing
- Blocks are enclosed code portions and may be executed at any moment
- Protocols and categories as extensions to the fundamental language

Don't worry if we're unfamiliar with these Objective-C features right now. As we continue reading this essay, you will discover more about them. If we are procedural programmers new to object-oriented notions, it may be helpful to see an object as a structure with related methods. This concept is not too distant from the truth, especially regarding runtime implementation.

In addition to offering the majority of abstractions and methods available in other object-oriented programming languages, Objective-C is a highly dynamic programming language, which is its most significant feature. It is dynamic in that it allows the behavior of an application to be decided

as it is running (i.e., during runtime) as opposed to being fixed when the application is developed. Thus, the dynamic nature of Objective-C liberates a program from limitations imposed during compilation and linking; it pushes the majority of symbol resolution responsibilities to runtime when the user is in control.

CLASSES AND OBJECTS

As with most other object-oriented programming languages, Objective-C classes enable data encapsulation and specify the operations performed on that data. An object is a class's instance during runtime. It holds a copy of the instance variables specified by its class and pointers to the class's methods. An object is created in a two-step method known as allocation and initialization.

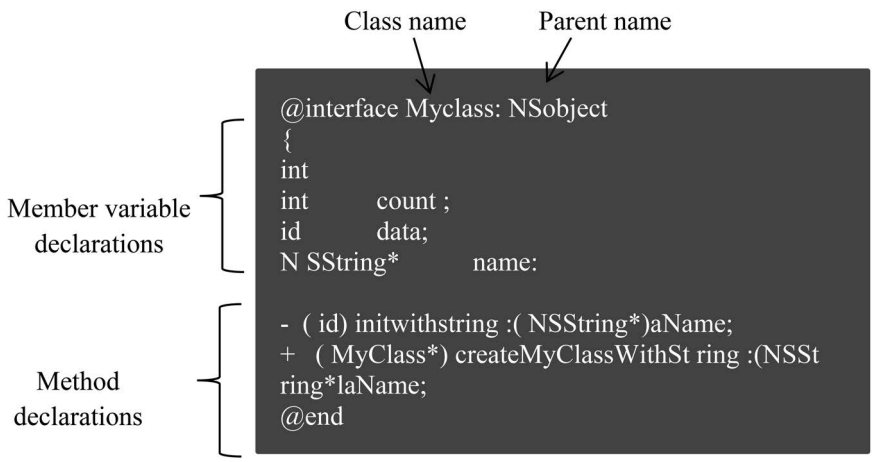
The Objective-C definition of a class involves two independent components: the interface and the implementation. The interface section includes the class declaration and specifies the class's public interface. Like C, we create header and source files to segregate public declarations from implementation details. (Other declarations are included in the implementation file if they are part of the programmatic interfaces but are intended to be private.) The filename extensions for these files are mentioned in the table below.

Extension	Source Type
.h	Files with headers. Header files include declarations for classes, types, functions, and constants.
.m	Files for implementation. This extension indicates that the file contains both Objective-C and C code. It is also known as a source file.
.mm	Files for implementation. This extension allows C++ code to be included in Objective-C and C code implementation files. Use this extension only if our Objective-C code refers to C++ classes or features.

When including header files in our source code, add a pound import (#import) directive as one of the first lines in a header or source file; a #import directive is similar to C's #include directive, but it ensures that the same file is never included more than once. Import the framework's umbrella header file in Objective-C, which has the same name as the framework, if we need to import most or all of the framework's header files. The following is the syntax for importing the (fictitious) Gizmo framework's header files:

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

The syntax for declaring a class called MyClass in Objective-C, which inherits from the base (or root) class, NSObject, is shown below. (A root class is one from which all other classes directly or indirectly derive.) The @interface compiler directive begins the class definition and concludes with the @end directive. The name of the parent class comes after the class name (and is separated by a colon). A class in Objective-C can only have one parent.



Declaration of members and method.

Declarations of properties and methods are written between @interface and @end. These declarations provide the class's public interface. ("Declared Properties and Accessor Methods" describes declared properties.) A semicolon is used after each property and method declaration. Place declarations for any custom methods, constants, or data types related to the class's public interface outside the @interface…@end block.

Similar syntax applies to class implementations. It starts with a @implementation directive followed by the class's name and concludes with a @ end directive. Method implementations exist in the middle. (Function implementations should not be placed inside the @implementation… @ end block.) Always import the interface file as one of the initial lines of code in an implementation.

```
#import "MyClass.h"

@implementation MyClass
- (id)initWithString:(NSString *)aName
```

```
{
    // here the code goes
}

+ (MyClass *)myClassWithString:(NSString *)aName
{
    // here the code goes
}
@end
```

Objective-C provides both dynamic and static typing for variables holding objects. The class name is included in the variable type declaration for statically typed variables. Instead, dynamically typed variables utilize the object's type id. In some cases, dynamically typed variables are employed. Dynamically typed variables are used in a collection object such as an array (where the exact types of the included objects are uncertain). Such variables provide for increased dynamism and flexibility in Objective-C programming.

This example displays variable declarations that are statically and dynamically typed:

```
MyClass *myObject1;  // Static-typing
id       myObject2;  // Dynamic-typing
NSString *user_Name;  // From our First iOS App
(static typing)
```

The initial declaration has an asterisk (*). Object references in Objective-C must always be pointers. Don't be concerned if this criterion does not make total sense to us. We don't have to be a pointer specialists to start Objective-C programming. Just remember to insert an asterisk in front of the variable names for statically typed object declarations. The id type implies a pointer.

METHODS AND COMMUNICATION

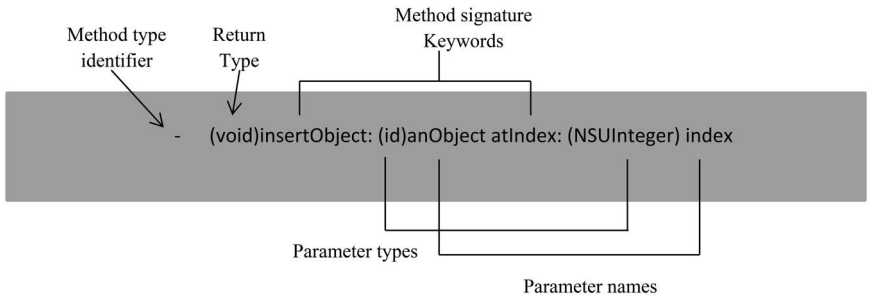
If we are new to object-oriented programming, it may be helpful to conceive of a method as an object-specific function. We invoke its method by sending a message to, or messaging, an object. In Objective-C, there are two types of methods: instance methods and class methods.

A method whose execution is restricted to a specific instance of a class is an instance method. In other words, we must construct an instance of

the class before using an instance method. Instance methods are the most prevalent method type.

The execution of a class method is limited to the method's class. An object instance doesn't need to be the recipient of a message.

The method's declaration includes the method type identifier, a return type, one or more signature keywords, and information on the argument type and name. Here is the method declaration for insertObject:atIndex: instance.



Declaration of object.

A minus sign precedes the declaration of instance methods (–), whereas class methods are indicated by a plus sign (+). The section titled "Class Methods" describes class methods in greater detail.

The actual name of a method (insertObject:atIndex:) is the concatenation of all signature keywords, including colons. The colon characters indicate that a parameter is present. In the preceding illustration, the method accepts two parameters. If a method has no parameters, the colon following the first and only signature keyword is omitted.

When calling a method, a message is sent to the object that implements the method. (Although "sending a message" is commonly used as a synonym for "calling a method," the actual sending is performed by the Objective-C runtime.) A message comprises the method's name and the parameter information required (properly conforming to type). All messages sent to an object are dispatched dynamically, making the polymorphic behavior of Objective-C classes possible. (Polymorphism refers to the capacity of various object types to respond to the same message.) Occasionally, the method invoked is implemented by a superclass of the receiving object's class.

For the runtime to dispatch a message, a message expression is necessary. A message expression encloses the message (along with any necessary parameters) within brackets ([and]) and, just inside the leftmost bracket, the object receiving the message. For instance, the following syntax is used to send the insertObject:atIndex: message to an object held by the myArray variable:

```
[myArray insertObject:anObject atIndex:0];
```

To avoid declaring multiple local variables to store interim results, we can nest message expressions in Objective-C. Each expression's return value is used as a parameter or as the receiving object of another message. To retrieve the values, we could, for instance, replace any of the variables used in the previous example with messages. Consequently, if we had another object named myAppObject with methods for accessing the array object and the object to insert into the array, you could have written the preceding example as follows:

```
[[myAppObject theArray] insertObject:[myAppObject
objectToInsert] atIndex:0];
```

For invoking accessor methods, Objective-C additionally supports a dot-notation syntax. Accessor methods get and set an object's state, which is critical to encapsulation, a fundamental aspect of all objects. Objects conceal or encapsulate their state and provide a standard interface for all instances to access that data. We may rewrite the preceding example using dot-notation syntax as follows:

```
[myAppObject.theArray insertObject:myAppObject.
objectToInsert atIndex:0];
```

For assignment, we may also use dot-notation syntax:

```
myAppObject.theArray = aNewArray;
```

This is merely another way of saying [myAppObject setTheArray: aNewArray];. A reference to a dynamically typed object (object of type id) is not used in a dot-notation expression.

We've already used dot syntax to assign a variable in our First iOS App:

```
self.user_Name = self.textField.text;
```

CLASS METHODS

Even though the preceding examples send messages to an instance of a class, it is also possible to send messages to the class itself. (A class is a runtime-generated object of type Class.) When communicating with a class, the method you specify must be a class method rather than an instance method. Class methods in C++ are comparable to static class methods.

Class methods are frequently used as factory methods to create new class instances or to access shared information associated with the class. The syntax for a class method declaration is identical to that of an instance method, except that the method type identifier is preceded by a plus sign (+) instead of a minus sign (–).

Following is an example of using a class method as a factory method for a class. In this instance, the array method is a class method on the NSArray and NSMutableArray classes that allocates, initializes, and returns a new class instance.

```
NSMutableArray *myArray = nil;  // nil is essentially
same as NULL

// Create new array and assign it to myArray variable.
myArray = [NSMutableArray array];
```

Properties and Accessor Methods Are Declared

The property is some data encapsulated or stored by an object in the broadest sense. It is either an attribute or a relationship to one or more other objects, such as a name or a color. The class of an object defines an interface that allows users of its objects to retrieve and modify the values of encapsulated properties. The methods that carry out these operations are referred to as accessor methods.

There are two categories of accessor methods, each of which must adhere to a naming convention. The "getter" accessor method that returns a property's value has the same name as the property. A "setter" accessor method has the form setPropertyName:, with the first letter of the property name capitalized. Properly named accessor methods are essential to several Cocoa and Cocoa Touch framework technologies, including key-value coding (KVC), a mechanism for indirectly accessing an object's properties via their names.

Declared properties provide a notational convenience for the declaration and implementation of accessor methods in Objective-C. In our First iOS App, the user_Name property was declared:

```
@property (nonatomic, copy) NSString *user_Name;
```

Declared properties avoid the need to implement getter and setter methods for each accessible property in the class. Instead, we use the property declaration to indicate the desired behavior. The compiler can then generate (or synthesis) real getter and setter methods based on that declaration. Declared properties decrease the amount of boilerplate code we have to write, making our code clearer and less prone to errors. To acquire and set items of an object's state, use defined properties or accessor methods.

In our class interface, we put property declarations with method declarations. Public properties are declared in the class header files, whereas private properties are declared in a class extension in the source file. Controller objects, such as delegates and view controllers, should typically have private properties.

The @property compiler directive is used in the basic property declaration, followed by the type information and the property name. Custom options can describe how the accessor methods operate, whether the property is a weak reference and whether it is read-only. Following the @property directive, the alternatives are in parentheses.

The lines of code below demonstrate a few additional property declarations:

```
@property (copy) MyModelObject *theObject;  // Copy
object during assignment.
@property (readonly) NSView *rootView;   // Declare
only getter method.
@property (weak) id delegate;          // Declare
delegate as weak reference
```

The compiler automatically synthesizes declared properties. When a property is synthesized, it generates accessor methods for it and a private instance variable that "backs" the property. The name of the instance variable is the same as the property's name but with an underscore prefix (_). In methods for object initialization and deallocation, our App should directly access an instance variable (rather than its property).

We can override auto synthesis and explicitly synthesize a property to give an instance variable a new name. In the class implementation, use the @synthesize compiler directive to instruct the compiler to build the accessor methods and the appropriately named instance variable.[†] As an example:

```
@synthesize enabled = _isEnabled;
```

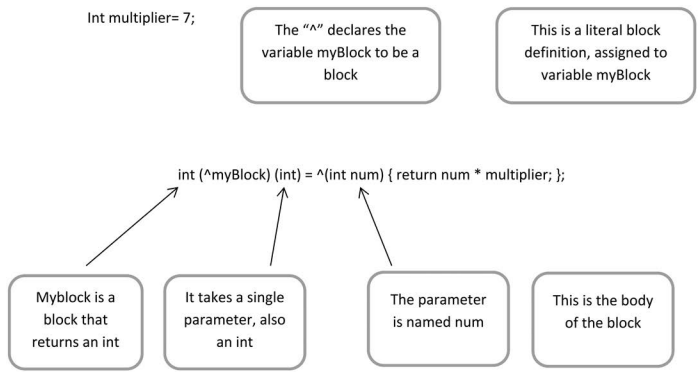
In addition, when we declare a property, we may give custom names for the accessor methods, which are commonly used to force the getter methods of Boolean properties to have a standard form, as illustrated here:

```
@property (assign, getter=isEnabled) BOOL enabled; //
Assign the new value, change name of getter method
```

BLOCKS

A block in Objective-C is an object that encapsulates a work unit or code segment that may be performed at any moment. Essentially, they are anonymous, portable functions that may be sent as inputs to methods and functions or returned from methods and functions. Blocks contain typed argument lists and may have an inferred or stated return type. Additionally, we may assign a block to a variable and then call it like we would a function.

A caret (^) is used to denote blocks syntactically. There are additional, well-known syntax norms for block arguments, return values, and body (the executed code). The following diagram illustrates the syntax for assigning a block to a variable.



Blocks.

[†] Write Objective-C Code

The block variable may then be accessed as if it were a function:

```
int result = myBlock(4); // result 28
```

A block shares local lexical scope info. If you implement a method and that method specifies a block, the block has access to the method's local variables and arguments (including stack variables) and functions and global variables, including instance variables. If a variable is defined with the __block modifier, its value is modified inside the block. As long as there is a reference to the block, the local variables continue even after the method or function inside the block has returned and its local scope has been deleted.

As arguments of a method or function, blocks might act as callbacks. When called, the method or function does some work and, at the proper times, calls back to the code that initiated it through the block to seek further information or acquire program-specific behavior. Blocks make it possible for the caller to give the callback code at invocation. Instead of encapsulating the necessary data in a "context" structure, blocks collect data from the same lexical scope as the host method or function. As the block code does not need to be implemented in a distinct method or function, the implementation code may be simplified and made easy to comprehend.

Many Objective-C framework methods use block arguments. For instance, the Foundation framework's NSNotificationCenter class specifies the following function with a block parameter:

```
- (id)addObserverForName:(NSString *)name object:(id)
obj queue:(NSOperationQueue *)queue usingBlock:(void
(^)(NSNotification *note))block
```

This technique installs a notification center observer (notifications are discussed in Streamline our App with Design Patterns). When a notice with the specified name is posted, the block is called to handle it.

```
opQ = [[NSOperationQueue alloc] init];
    [[NSNotificationCenter defaultCenter] addObserverF
orName:@"CustomOperationCompleted"
             object:nil queue:opQ
        usingBlock:^(NSNotification *notif) {
        // handle notification
    }];
```

PROTOCOLS AND CATEGORIES

A protocol specifies methods that may be implemented by any class, even if those classes do not share a superclass. Protocol methods create behavior that is agnostic to specific classes. Protocols merely specify an interface that other classes must implement. When your class implements a protocol's methods, it is said to conform to that protocol.

A protocol specifies a set of methods that forms a contract between objects without needing them to be instances of a specific class. This contract allows these items to communicate with one another. One object wants to inform another object about the events it is experiencing, or it may seek guidance on these occurrences.

The UIApplication class implements an application's needed functionality. Instead of requiring you to subclass UIApplication to get basic messages on the current status of the application, the UIApplication class provides these notifications by invoking particular methods of the delegate object it has allocated. An object that implements the UIApplicationDelegate protocol's methods may receive these alerts and respond appropriately.

In the interface block, you indicate that your class adheres to or adopts a protocol by enclosing the protocol's name in angle brackets (<…>) following the name of the class from which our class inherits. In our First iOS App, you adopted the UITextFieldDelegate protocol in the following line of code:

```
@interface HelloEveryoneViewController :
UIViewController <UITextFieldDelegate> {
```

We are not required to declare the protocol methods that we implement.

A protocol's declaration resembles that of a class interface, with the distinction that protocols do not have a parent class and do not have instance variables (although they can declare properties). The following is a basic protocol declaration with only one method:

```
@protocol MyProtocol
- (void)myProtocolMethod;
@end
```

Adopting a protocol for many delegate protocols is as simple as implementing the techniques described by that protocol. Some protocols need

us to say that you support the protocol explicitly, and protocols might have both required and optional ways.

When we start looking through the header files of the Objective-C frameworks, we ll come across something like this:

```
@interface NSDate (NSDateCreation)
```

This line defines a category by surrounding the category's name in parentheses, per syntax standard. A category is an Objective-C language feature that allows us to expand the interface of a class without subclassing. The category's methods become part of the class type (within the program's scope) and are inherited by all subclasses of the class. We may send a message to any class instance (or its subclasses) to activate a category-defined method.

Categories are used to organize similar method declarations inside a header file. Even distinct category definitions are included in separate header files. These strategies are used throughout the frameworks' header files for clarity.

We may also use a class extension, which is an anonymous category, to specify private properties and methods in the implementation (.m) file. The only difference between a class extension and a category is the absence of text between the parenthesis. Here is an example of a typical class extension:

```
@interface MyAppDelegate ()
@property (strong) MyDataObject *data;
@end
```

Types and Coding Strategies Are Defined

Several words in Objective-C cannot be used as variable names because they are reserved for particular uses. These words are compiler instructions with at-signs (@) prefixes, such as @interface and @end. Other restricted phrases include specified types and their associated literals. Objective-C employs a variety of types and literals that are not included in ANSI C. In some instances, these types and literals substitute their ANSI C equivalents. The following table lists some of the essential literal types and the permitted literals for each.

Type	Description and Literal
id	The kind of dynamic object. For both dynamically and statically typed objects, the negative literal is nil.
Class	The kind of dynamic class. Its negative literal is Nil.
SEL	A selector's data type (typedef); this data type reflects a method signature at runtime. It is inverse literal NULL.
BOOL	A Boolean expression. YES and NO are the literal values.

These specified types and literals are frequently used in error-checking and control-flow programs. We may test the proper literal in our program's control-flow statements to determine how to continue. As an example:

```
NSDate *dateOfHire = [employee dateOfHire];
if (dateOfHire != nil) {
    // handle this
}
```

To summarize, if the object indicating the date of hire is not nil if it is a real object, then the reasoning proceeds on a certain path. Here's a quick technique to perform the same branching:

```
NSDate *dateOfHire = [employee dateOfHire];
if (dateOfHire) {
    // handle this
}
```

We can trim these lines of code even further (assuming we don't need a reference to the dateOfHire object):

```
if ([employee dateOfHire]) {
    // handle this
}
```

We handle Boolean values in a similar manner. The isEqual: method in this example returns a Boolean value.

```
BOOL equal = [objectA isEqual:objectB];
if (equal == YES) {
    // handle this
}
```

Shorten this code similarly to the code that checks for the lack or existence of nil.

In Objective-C, sending a message to nil has no negative consequences. There is no consequence other than the runtime returning null if the method should return an object. Return values from messages delivered to nil are guaranteed to function as long as they are of the object type.

The self and super are also key reserved concepts in Objective-C. The first word, self, is a local variable that may refer to the current object inside a message implementation; it is analogous to this in C++. The reserved term super may be substituted for self, but only as of the recipient of a message expression. If we send a message to self, the runtime first searches for the method implementation in the current object's class; it searches for it in its superclass if it cannot find it there. When a message is sent to a super, the runtime first searches for the method implementation in the superclass.

Both self and super is mostly used for transmitting messages. When the self class implements the method to call, we send a message to the self. For instance:

```
[self doSomeWork];
```

In dot notation, the self is also called the accessor method created by a defined property. As an example:

```
NSString *theName = self.name;
```

Messages to super are frequently sent in overrides (reimplementations) of methods inherited from a superclass. The method invoked in this scenario has the same signature as the method overwritten.

CREATE THE VIDEO App

Create the code for an introductory video player App.

Set the App's Audio Behavior

At the app level, the audio session manages audio behavior. Explore the AVAudioSession class in further detail.

We will utilize AVAudioSessionCategoryPlayback for this example. Even when the Ring/Silence switch is silent, and the screen is locked, this plays audio. To maintain simplicity, we will place this code in the App Delegate.

Open the App Delegate implementation file in our project (AppDelegate.m).

Add code to the didFinishLaunchingWithOptions function to set the audio session type. Ensure that the AVFoundation framework is imported.

The following code is required to guarantee that audio is played when expected. Without setting this code, we cannot hear the video while the mute button is used.

For simplicity in the example, we've placed this in the App delegate.

```
//  AppDelegate.m
//  Simple VideoPlayback

#import "AppDelegate.h"
#import <AVFoundation/AVFoundation.h>

@interface AppDelegate ()

@end

@implementation AppDelegate

- (BOOL)application:(UIApplication *)application didFi
nishLaunchingWithOptions:(NSDictionary *)launchOptions
{
// Override point for the customization after
application run

NSError *categoryError = nil;
BOOL success = [[AVAudioSession sharedInstance] setCat
egory:AVAudioSessionCategoryPlayback
error:&categoryError];

if (!success)
{
        NSLog(@"AppDelegate Debug - Error setting
AVAudioSession category.  Because of this in
Objective-C,  there may be no sound. '%@'",
categoryError);
}
return YES;
}
```

```
- (void)applicationWillResignActive:(UIApplication *)
application {
// Sent when the application is about to move in
Objective-C  from the active to an inactive state.
This can occur for certain temporary interruptions
(such as an incoming phone call or SMS message) or
when the user quits the application and begins the
transition to the background state.
// Use this method in Objective-C to pause ongoing
tasks, disable timers, and throttle down OpenGL ES
frame rates. Games should use this method in
Objective-C to pause the game.
}

- (void)applicationDidEnterBackground:(UIApplication
*)application {
// Use this method to release the shared resources,
save user data, invalidate timers, and store enough
application state information to restore your
application to its current state if it is terminated
later.
// If our application supports background execution,
this method is called instead of
applicationWillTerminate: when a user quits.
}

- (void)applicationWillEnterForeground:(UIApplication
*)application {
// Called as part of the transition from the
background in Objective-C to the inactive state. We
can undo many of the changes made on entering the
background.
}

- (void)applicationDidBecomeActive:(UIApplication *)
application {
// Restart any paused (or not yet started) tasks while
the application was inactive. If the application in
Objective-C was previously in the background, refresh
the user interface optionally.
}
```

```
- (void)applicationWillTerminate:(UIApplication *)
application {
// Called when application is about to terminate. Save
data if appropriate. See also applicationDidEnter
Background:.
}

@end
```

Build View Controller Class Declaration

Include the Brightcove SDK in the View Controller class.

Import the Brightcove Player SDK Header File into the Program

For the Brightcove Player SDK, add the following import statement:

```
@import BrightcovePlayerSDK;
```

Look at the Code

The View Controller header has now been finished. Here is the complete code:

```
//  ViewController.h
//  Simple VideoPlayback

#import <UIKit/UIKit.h>
@import BrightcovePlayerSDK;

@interface ViewController : UIViewController

@end
```

Construct the View Controller Implementation in Objective-C

In Objective-C to play a video from the Brightcove catalog service, update the View Controller implementation.

Customize the Project to Reflect Our Values

To access our Video Cloud account, enter the token and playlist id values.

This example displays a video returned using the Brightcove Playback API, the most recent and recommended API for retrieving material from our Video Cloud library.

Add our values for the below the #import directives:

- This line specifies Our Brightcove Playback API Policy Key.
- Our Video Cloud Account ID is defined here.
- Specifies the Video Cloud Video ID.

```
// ** Customize these values with our own account
information **
static NSString * const
kViewControllerPlaybackServicePolicyKey = @"our policy
key";
static NSString * const kViewControllerAccountID =
@"our account id";
static NSString * const kViewControllerVideoID = @"our
video id";
```

Declare Properties

Add the following class declarations to the ViewController interface section:

- Brightcove delegates are added to the UI. This enables our App to respond to video playback events.
- The BCOVPlaybackService class defines methods for getting data from the Playback API asynchronous.
- The BCOVPlaybackController class defines methods for controlling playback capability.
- Defines the Brightcove UI controllers' player view.
- This property defines the video container view.

```
@interface ViewController ()
<BCOVPlaybackControllerDelegate>

@property (nonatomic, strong) BCOVPlaybackService
*playbackService;
@property (nonatomic, strong)
id<BCOVPlaybackController> playbackController;
@property (nonatomic) BCOVPUIPlayerView *playerView;
```

```
@property (nonatomic, weak) IBOutlet UIView
*videoContainer;

@end
```

DEFINE INITIALIZATION METHOD

In the ViewController implementation section, write an init function that calls the setup method defined in the next step.

```
@implementation ViewController
#pragma mark the Setup Methods

- (instancetype)initWithCoder:(NSCoder *)coder
{
self = [super initWithCoder:coder];
if (self)
{
        [self setup];
}
return self;
}
```

Setup Player

- Create a setup function underneath the init function invoked when the App loads.
- Creates a playback controller using the shared manager. The BCOVPlayerSDKManager class in Objective-C is a singleton that allows us to build additional SDK-related objects.
- Optional: We must transmit our Video Cloud Account ID to Video Cloud Analytics if you override the BCOVVideo class or do not utilize the Brightcove player and playback service or catalog. You can now access data for this App in Video Cloud Analytics.

```
- (void)setup
{
_playbackController = [BCOVPlayerSDKManager.
sharedManager createPlaybackController];

_playbackController.analytics.account =
kViewControllerAccountID; // this is Optional
```

```
_playbackController.delegate = self;
_playbackController.autoAdvance = YES;
_playbackController.autoPlay = YES;

_playbackService = [[BCOVPlaybackService alloc] initWi
thAccountId:kViewControllerAccountID policyKey:kViewCo
ntrollerPlaybackServicePolicyKey];
}
```

Configure Player

Do the following in the viewDidLoad function:

- Using the usual VOD structure, create and configure the Brightcove player controls.
- The player view is added as a subview to the main view.
- Turn off the auto-resize mask.
- To establish dynamic constraints for the player view, use Auto Layout.
- The player view is assigned to the associated global variable.
- The player view is linked to the playback controller.
- This method invokes the requestContentFromPlaybackService function, which we will write in the following step.

```
- (void)viewDidLoad {
    [super viewDidLoad];
    // Do any additional setup after loading view,
typically from nib.

    // Setup our player view. Create with a standard
VOD layout in Objective-C.
    BCOVPUIPlayerView *playerView =
[[BCOVPUIPlayerView alloc]
initWithPlaybackController:self.playbackController
options:nil controlsView:[BCOVPUIBasicControlView
basicControlViewWithVODLayout] ];

    _videoContainer addSubview:playerView];
    playerView.
translatesAutoresizingMaskIntoConstraints = NO;
```

```
    [NSLayoutConstraint activateConstraints:@[
        [playerView.topAnchor
constraintEqualToAnchor:_videoContainer.topAnchor],
        [playerView.rightAnchor
constraintEqualToAnchor:_videoContainer.rightAnchor],
        [playerView.leftAnchor
constraintEqualToAnchor:_videoContainer.leftAnchor],
        [playerView.bottomAnchor
constraintEqualToAnchor:_videoContainer.bottomAnchor],
    ]];
    _playerView = playerView;

    // Associate playerView with playback controller.
    _playerView.playbackController =
_playbackController;

    [self requestContentFromPlaybackService];
  }
```

Use the Brightcove Library to Request Material

You must first request a video from our Video Cloud collection to play video content.

Create a method called requestContentFromPlaybackService underneath the viewDidLoad function as follows:

```
- (void)requestContentFromPlaybackService
{
[self.playbackService findVideoWithVideoID:kViewContro
llerVideoID parameters:nil completion:^(BCOVVideo
*video, NSDictionary *jsonResponse, NSError *error) {

    if (video)
    {
        [self.playbackController setVideos:@[ video
]];
    }
    else
    {
        NSLog(@"ViewController Debug - The Error
retrieving video: '%@'", error);
    }

}];
}
```

Look at the Code

The implementation of the View Controller is now complete. Here is the complete code:

```
//  ViewController.m
//  Simple-Video-Playback

#import "ViewController.h"

// ** Customize these values with our own account
information **
static NSString * const
kViewControllerPlaybackServicePolicyKey = @"our policy
key";
static NSString * const kViewControllerAccountID =
@"our account id";
static NSString * const kViewControllerVideoID = @"our
video id";

@interface ViewController ()
<BCOVPlaybackControllerDelegate>

@property (nonatomic, strong) BCOVPlaybackService
*playbackService;
@property (nonatomic, strong)
id<BCOVPlaybackController> playbackController;
@property (nonatomic) BCOVPUIPlayerView *playerView;
@property (nonatomic, weak) IBOutlet UIView
*videoContainer;

@end

@implementation ViewController
#pragma mark Setup Methods

- (instancetype)initWithCoder:(NSCoder *)coder
{
self = [super initWithCoder:coder];
if (self)
{
        [self setup];
}
```

```
return self;
}

- (void)setup
{
_playbackController = [BCOVPlayerSDKManager.
sharedManager createPlaybackController];

_playbackController.analytics.account =
kViewControllerAccountID; // this is optional

_playbackController.delegate = self;
_playbackController.autoAdvance = YES;
_playbackController.autoPlay = YES;

_playbackService = [[BCOVPlaybackService alloc] initWi
thAccountId:kViewControllerAccountID policyKey:kViewCo
ntrollerPlaybackServicePolicyKey];
}

- (void)viewDidLoad {
[super viewDidLoad];
// Do any additional setup after loading view,
typically from nib.

// Set up our player view. Create with a standard VOD
layout.
BCOVPUIPlayerView *playerView = [[BCOVPUIPlayerView
alloc] initWithPlaybackController:self.
playbackController options:nil controlsView:[BCOVPUIBa
sicControlView basicControlViewWithVODLayout] ];

[_videoContainer addSubview:playerView];
  playerView.translatesAutoresizingMaskIntoConstraints
= NO;
  [NSLayoutConstraint activateConstraints:@[
    [playerView.topAnchor constraintEqualToAnchor:_
videoContainer.topAnchor],
    [playerView.rightAnchor constraintEqualToAnchor:_
videoContainer.rightAnchor],
    [playerView.leftAnchor constraintEqualToAnchor:_
videoContainer.leftAnchor],
```

```
    [playerView.bottomAnchor constraintEqualToAnchor:_
videoContainer.bottomAnchor],
]];
_playerView = playerView;

// Associate playerView with playback controller.
_playerView.playbackController = _playbackController;

[self requestContentFromPlaybackService];
}

- (void)requestContentFromPlaybackService
{
[self.playbackService findVideoWithVideoID:kViewContro
llerVideoID parameters:nil completion:^(BCOVVideo
*video, NSDictionary *jsonResponse, NSError *error) {

    if (video)
    {
        [self.playbackController setVideos:@[ video
]];
    }
    else
    {
        NSLog(@"ViewController Debug - The Error
retrieving video: '%@'", error);
    }

}];
}

@end
```

In this chapter, we covered Functional Programming with writing methods and a sample of the project in Objective-C.

NOTE

1. Functional Programming in Objective-C

BIBLIOGRAPHY

1. Functional Programming in Objective-C – https://bou.io/FunctionalProgrammingInObjectiveC.html, accessed on May 11, 2022.
2. Objective-C Functions – https://www.tutorialspoint.com/objective_c/objective_c_functions.htm, accessed on May 11, 2022.

3. Higher-Order Functions in Objective-C – https://betterprogramming.pub/higher-order-functions-in-objective-c-850f6c9–0de30, accessed on May 11, 2022.
4. Functions In Objective-C – https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwi-tauczPT6AhUCRmwGHVNQDuUQFnoECBwQAQ&url=https%3A%2F%2Fblog.devgenius.io%2Ffunctions-in-objective-c-1282145253f2&usg=AOvVaw3sPselGfVcc4xKYk3WfGaS, accessed on May 11, 2022.
5. Essence of object Functional Programming practical potential of Scala – https://blog.codecentric.de/essence-of-object-functional-programming-practical-potential-of-scala#:~:text=Object%2Dfunctional%20programming%20is%20a,and%20principles%20to%20this%20end, accessed on May 12, 2022.
6. Object-Oriented Functional Programming – https://academy.realm.io/posts/altconf-saul-mora-object-orientated-functional-programming/, accessed on May 12, 2022.
7. Difference between Functional Programming and Object Oriented Programming – https://www.geeksforgeeks.org/difference-between-functional-programming-and-object-oriented-programming/, accessed on May 12, 2022.
8. Functional Programming in Objective-C – https://bou.io/FunctionalProgrammingInObjectiveC.html, accessed on May 12, 2022.
9. Write Objective-C Code – https://developer.apple.com/library/archive/referencelibrary/GettingStarted/RoadMapiOS-Legacy/chapters/WriteObjective-CCode/WriteObjective-CCode/WriteObjective-CCode.html#:~:text=Objective%2DC%20Is%20a%20Superset%20of%20the%20C%20Language&text=As%20a%20superset%20of%20the,..else%20and%20for%20statements, accessed on May 12, 2022.
10. Objective-C Basic Syntax – https://www.tutorialspoint.com/objective_c/objective_c_basic_syntax.htm, accessed on May 12, 2022.
11. Objective-C Hello World Tutorial – https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwjuteiEzfT6AhWBXmwGHTfsAegQFnoECB8QAQ&url=https%3A%2F%2Fwww.digitalocean.com%2Fcommunity%2Ftutorials%2Fobjective-c-hello-world-tutorial&usg=AOvVaw3b7aBcZTghSGFmBvL7pZ_t, accessed on May 12, 2022.
12. Programming in Objective-C: Creating Your First Program – https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwjuteiEzfT6AhWBXmwGHTfsAegQFnoECCIQAQ&url=https%3A%2F%2Fwww.informit.com%2Farticles%2Farticle.aspx%3Fp%3D2159356&usg=AOvVaw1saEKgSLXoLUdgeJHHQ31Z, accessed on May 12, 2022.
13. Write Objective-C Code – https://developer.apple.com/library/archive/referencelibrary/GettingStarted/RoadMapiOS-Legacy/chapters/WriteObjective-CCode/WriteObjective-CCode/WriteObjective-CCode.html, accessed on May 12, 2022.

CHAPTER 5

Code Management

IN THIS CHAPTER

- Frameworks
- Memory Management
- System Design

In the previous chapter, we covered functional programming, and in this chapter, we will discuss code management with its relevant examples.

Allocate memory for each object used by our program. It must be deallocated when the object is no longer needed so that our application utilizes memory as effectively as possible. To prevent memory leaks and references to nonexistent objects, it is essential to comprehend Objective-memory C's management environment.

Unlike C#, Objective-C does not employ trash collection. Instead, it employs a reference-counting environment that monitors the number of instances of an object. As long as at least one reference to the object exists, the Objective-C runtime guarantees that the object is in memory. Nevertheless, if there are no longer any references to the object, the runtime is permitted to release it and utilize the memory for something else. If we attempt to access an object after it has been released, your application may probably crash.

DOI: 10.1201/9781003357506-5

In Objective-C, there are two mutually incompatible methods for managing object references:

- Send ways to manually add or reduce the number of references to an object.
- Allow the new automated reference counting (ARC) mechanism in Xcode 4.2 (and later) to perform the job for you.
- Although ARC is the recommended method for managing memory in new programs, it is still vital to understand what is going on below the hood. The first section of this chapter demonstrates how to manually track object references, followed by a discussion of the practical ramifications of ARC.

A framework is conceptually just a mechanism to modularize compiled code. Frameworks are the best approach to sharing code reused inside and outside your company. They make maintaining library dependencies simpler than managing a binary package and a global header.

A framework is a specific kind of macOS bundle (and macOS bundles are simply files) intended for usage by developers. Often, a framework bundle includes development resources (typically statically-built libraries and interface headers), but it may also include graphics, storyboards, xibs, and property list files.

A compiler may extract built binaries from a framework and link them during build or run time. The Finder presents bundles with a .framework extension as regular folders for easy review by a developer.

Due to these unique characteristics, framework bundles need precise directory layouts and file placements to function successfully, especially for their most frequent use, binary code distribution.

WHY MUST WE PERFORM THIS?

There are several reasons why we would want to design a framework and a few reasons why we must. Frameworks are the ideal approach for us and our developers to provide built libraries for usage outside our business. They may also make the structure of a large application much easier to maintain since we can divide the application into many targets and manage each project individually. Fundamentally, there are three fundamental reasons why frameworks exist:

- Modularity: Remove and replace independent portions of your source without worrying about weird compiler errors or linker issues. Work on discrete portions of our codebase.
- Reuse: Group-related processes so that they are reused across our application or applications.
- Encapsulation: Develop portions of your codebase to be wholly independent and interference-free.

We are likely already familiar with these concepts. However, due to the intricacies of establishing a framework, many individuals avoid doing so.

However, if we're designing iOS applications that are more complex than the most fundamental, we should consider frameworks.

ANATOMY OF A FRAMEWORK

There are several methods to build and package a framework bundle, and our decisions should be based on how you see the framework being used in production. If we've never done anything like this before, it's good to familiarize ourselves with Apple's instructions on framework bundles before proceeding.

Typically, a structure includes the following:

- Compilation of binary code: A well-constructed framework should have numerous binaries for various processor architectures.
- Interface header: Since this is Objective-C, we will need to supply headers for each class in our framework.
- Different resources: Sometimes, frameworks are utilized to distribute additional resources. There are always better methods to share resources that belong to our code, such as a dedicated resources package. Thus we would not recommend doing this.

STATIC AND DYNAMIC FRAMEWORKS

There are two types of frameworks: static and dynamic. The distinction between the two is precisely as it sounds: static frameworks are connected during the construction phase, while dynamic frameworks are linked during the compilation phase.

Apple's frameworks are all dynamically linked, whereas yours may be either. Before iOS 8, dynamic linking of frameworks is only supported on macOS; however, it is now supported on iOS.

Static frameworks might increase your application's disc size and build time since the complete binary is transferred and linked during the build. Dynamic frameworks only employ the necessary portions of the framework at build time and execute linking during runtime, possibly increasing your App's start time but decreasing its total disc size.

A further benefit of dynamic frameworks is that frameworks are updated without rebuilding the application since linking occurs on every startup.

Dynamic frameworks are unquestionably the more current approach, but there are excellent reasons to utilize static frameworks, particularly for code distributed outside the enterprise.

ARCHITECTURES AND SLICING OF PROCESSORS

Typically, a framework's binaries include many copies of the same code tailored to various processor architectures. These parts are called slices since they are combined into a single binary. To construct an appropriate framework, we must include slices for each version of the ARM instruction set used by iOS devices that we want to support and a slice for the specific x86 version of iOS used by the iPhone simulator.

DEVELOPING A DYNAMIC STRUCTURE

Creating a dynamic framework using Xcode is far less complicated than we may believe. We will create the ExampleKit framework in this example, which includes the EKObject data structure and the EKSession processing object. We can locate both classes in this location, but we shouldn't interact with them just yet.

Create a new Xcode project to begin. Select "Cocoa Touch Framework" and name our project ExampleKit. Building a git repository for our project is a smart idea.

SETTING UP OUR PROJECT

Let's examine what Apple's template currently gives us before we continue.

- We already have an umbrella header with explicit instructions for importing public headers.
- All required construction stages are currently in place.
- Setting code signing to "do not code sign."

As we'll see when we construct a static framework, these are advantageous qualities. Apple has completed the majority of the challenging tasks for us. We simply need to specify a deployment target, the minimal version of iOS we want to support.

DEVELOPING OUR CODE

If we've previously created an iOS App target, we're likely acquainted with "target membership." We know that resources are included in the App's "copy resources" build phase, implementation files must be included in the "compile sources" build phase, and header files are not typically specified in an App's build phases.

Add both classes to your project and ensure that they have been copied to the source directory. After doing so, ensure that both.m files are members of our framework's "ExampleKit" target.

Headers are also part of the construction process and must be target members when constructing a framework. We may have previously seen that the target membership option is accessible in the file inspector.h files. Add them as additional members of the target.

ACCESS CONTROL

When we uploaded the header files to the target, we were also presented with an access control picker. Ensure both header files are set to "Public" as opposed to "Project" or "Private."

These names are somewhat deceptive; all headings will be accessible to our customers and searchable in the finder, but headers must be expressly made public to be imported into the umbrella header. In general, we should place headers that our customers should use in "Public" and headers that are particular to our library's implementation in "Project" and "Private." They are not hidden but won't appear in code completion, won't be imported in our umbrella header, and are commonly recognized as not being used outside the framework. (This is something Swift permits that Objective-C does not – real file and project privacy.)

UMBRELLA HEADER

Regarding the umbrella header, we must return and make our newly created class interfaces visible. This allows them to be imported with a single sentence. In the ExampleKit.h file, import them as follows:

```
#import <ExampleKit/EKObject.h>
#import <ExampleKit/EKSession.h>
```

UNIVERSAL SUPPORT

We're using an Xcode template explicitly intended for developing dynamic frameworks, so most of the setting has already been completed. Compile our code once to ensure that everything is functioning properly before continuing.

Remember when we were discussing the business of slicing? Xcode will only construct our framework for the specified platform and related slices. Thus, if we develop an iPhone simulator, we will get a framework with an x86 slice but no ARM slice. If we develop for iPhones, we will get a framework with arm slice(s). However, simulator support is not included.

We will need to construct a script that compiles for *both* platforms and *all* viable slices, combines the binaries generated by each and generates a framework using the structure from one of the two single-platform builds.

```
set -e
set +u

if [[ $SCRIPT_RUNNING ]]
then
exit 0
fi
set -u
export SCRIPT_RUNNING=1

# Environment Variables
TARGET_NAME=${PROJECT_NAME}
OUTPUT_DIR=${PROJECT_DIR}/Release

# Encapsulate the Xcode Build Process
function build_dynamic_framework {

    xcrun xcodebuild -project "${PROJECT_FILE_PATH}" \
    -target "${PROJECT_NAME}" \
    -configuration "${CONFIGURATION}" \
    -sdk "${1}" \
    ONLY_ACTIVE_ARCH=NO \
    BUILD_DIR="${BUILD_DIR}" \
    OBJROOT="${OBJROOT}" \
    BUILD_ROOT="${BUILD_ROOT}" \
    SYMROOT="${SYMROOT}" $ACTION

}
```

```
# Encapsulate the Lipo
function merge_binaries {

    xcrun lipo -create "${1}" "${2}" -output "${3}"

}

# 1 - Get SDK to determine platform (iphoneos or
iphonesimulator)
if [[ "$SDK_NAME" =~ ([A-Za-z]+) ]]; then
SDK_PLATFORM=${BASH_REMATCH[1]}
else
echo "Could not find platform name from SDK_NAME:
$SDK_NAME"
exit 1
fi

# 2 - Get the Opposite Platform (iphonesimulator -->
iphoneos, iphoneos --> iphonesimulator)
if [ "$SDK_PLATFORM" == "iphoneos" ]; then
OTHER_PLATFORM=iphonesimulator
else
OTHER_PLATFORM=iphoneos
fi

# 3 - Get build directories
CURRENT_DIR=${BUILD_DIR}/${CONFIGURATION}-${SDK_
PLATFORM}
OTHER_DIR=${BUILD_DIR}/${CONFIGURATION}-${OTHER_
PLATFORM}

# 4 - Build the Opposite Platform
build_dynamic_framework "${OTHER_PLATFORM}"

# 5 - Copy Framework Structure
rm -rf "${OUTPUT_DIR}"
mkdir -p "${OUTPUT_DIR}"
cp -R "${BUILD_DIR}/${CONFIGURATION}-${SDK_
PLATFORM}/${PROJECT_NAME}.framework" "${OUTPUT_
DIR}/${PROJECT_NAME}.framework"
```

```
# 6 - Merge Into /Release
merge_binaries "${CURRENT_DIR}/${PROJECT_NAME}.
framework/${PROJECT_NAME}" "${OTHER_DIR}/${PROJECT_
NAME}.framework/${PROJECT_NAME}" "${OUTPUT_
DIR}/${PROJECT_NAME}.framework/${PROJECT_NAME}"
```

Instead of simply naively copying and pasting this script, ensure we comprehend its purpose. You may need to adjust it for future Xcode releases and our project's particular settings. It may be advantageous to add the script as a separate file rather than entering it straight into Xcode so that we can more easily manage it using version control.

This step may not be necessary for every framework build throughout the development process since it might lengthen the duration of the build. Several methods exist for separating the platform binding script:

- We might manually activate and disable it in the build steps of our target.
- We might construct our script only to execute when the RELASE setting is used.
- Instead of adding the script to our main target's build phases, we might establish a new aggregate target, connect our framework target as a dependency, and add the script to that target's build phases.

The first choice is the simplest. Precede our opponent's construction stages as the final phase. Then, just construct our target for "Generic iOS Device," and we're done; however, we will need to deactivate and re-enable the script as required.

The second alternative is similarly rather straightforward. Simply enclose our script in an if statement and determine if the build configuration is configured for release:

```
if [ $CONFIGURATION == Release ]; then
# script-here
fi
```

My preferred option is the third one. Create a new aggregate target with the name "Framework." Then, add our dynamic framework target as a dependency and instead add our script to the new "Framework" target. Then, construct the "Framework" target for "Generic iOS Device" when

building a universal binary, and only the framework target when building a binary containing slices exclusively for the presently chosen platform.

UTILIZING OUR DYNAMIC FRAMEWORK

Our dynamic framework differs somewhat from using Apple's OS-integrated dynamically linked frameworks. Because we're supplying the binary as part of our App, we must embed it, so the runtime knows to load and link it from inside the App upon startup.

DEVELOPING A STATIC FRAMEWORK

Constructing a static framework is more complex than building a dynamic framework since Xcode does not provide a suitable template. Due to this, we must begin with Xcode's static library template and add scripting to package the static library as a framework appropriately.

SETTING UP OUR PROJECT

Create a new Xcode Project and choose the "Cocoa Touch Static Library" template this time. Again, give this project the name "ExampleKit" and create a git repository beside it.

We will see some variations if we construct the dynamic framework before reaching this point. EmampleKit.h does not seem to be intended as an umbrella header. It defines an interface for a class named ExampleKit and includes a .m file for implementing the class.

Start by addressing this issue: remove the interface definition from ExampleKit.h and delete ExampleKit.m.

DEVELOPING OUR CODE

As in the previous example, our static framework will consist of two classes: EKObject and EKSession. Both are offered here. Add all four files to your project and ensure that EKObject.m and EKSession.m are included in your ExampleKit static library target.

ACCESS CONTROL

Previously, we could apply access control to each header by adding it to the target and selecting its access control level in the file and identity inspector. Since this template was not intended to be a framework, such functionality is not included. We must manually add a headers phase.

Click the plus sign (+) in the upper left corner of the "Build Phases" section of our target's settings to add a new "header phase."

Once this is complete, we will be able to add each heading as a member to the target and set a level of access control. Ensure that EKSession.h and EKObject.h are "public," but exclude ExampleKit.h from the target's membership.

UMBRELLA HEADER

Similarly to the dynamic framework, we will expose our headers in the umbrella header to import them all at once. Proceed to ExampleKit.h and import them as follows:

```
#import <ExampleKit/EKObject.h>
#import <ExampleKit/EKSession.h>
```

Construct our "ExampleKit" target and ensure it builds without any issues.

PACKAGING

Because we're using a template intended for a static library, we'll need to modify our target to generate a static framework. Our template does not provide the required directory structure and symbolic links, so we'll need to modify our target's build settings and write a script to do some hard work.

MODIFY BUILD SETTINGS TO SUPPORT STATIC FRAMEWORKS

- Change the Public Headers Search Path option to include/$(PROJECT NAME).
- Change setting for Dead Code Stripping to NO.
- Change the Style of the Strip to Non-Global Symbols.

Rebuild our static target, then right-click on "libExampleKit.a" and choose "Show in Finder." Our static library has been condensed into a single bundle and some headers. However, there is currently no unified framework.

MODULE SUPPORT

The dynamic framework template automatically generates a module map at build time and installs it in the corresponding directory. Clang Modules

are a superior method of importing files than #include (or #import, which is simply #include without the duplication) and are required if we want your framework to be compatible with Swift projects. Create a new empty file with the name "module.modulemap" It requires no target membership.

```
framework module ExampleKit {
    umbrella header "ExampleKit.h"

    export *
    module * { export * }
}
```

CREATING THE BUNDLE STRUCTURE

Add the following script as the last build step for our static library target:

```
set -e
export FRAMEWORK_LOCN="${BUILT_PRODUCTS_
DIR}/${PRODUCT_NAME}.framework"

# Create Directory for the Actual Headers Location for
Version A
mkdir -p "${FRAMEWORK_LOCN}/Versions/A/Headers"

# Symbolically Link Headers to their Parse Directory
ln -sfh A "${FRAMEWORK_LOCN}/Versions/Current"
ln -sfh Versions/Current/Headers "${FRAMEWORK_LOCN}/
Headers"
ln -sfh "Versions/Current/${PRODUCT_NAME}" \
"${FRAMEWORK_LOCN}/${PRODUCT_NAME}"

# Copy the Public Headers Into Their Directory
cp -a "${TARGET_BUILD_DIR}/${PUBLIC_HEADERS_FOLDER_
PATH}/" \
"${FRAMEWORK_LOCN}/Versions/A/Headers"

# Create the Module Directory
mkdir -p "${FRAMEWORK_LOCN}/Modules"

# Copy the Module Map
cp -f "${SRCROOT}/${PRODUCT_NAME}/module.modulemap"
"${FRAMEWORK_LOCN}/Modules/"
```

Now, regenerate our static library target for "Generic iOS Device." Display our libExampleKit.a product in the finder and examine it! We will now note that the "ExampleKit.framework" is now included in addition to the regular build products. Double-clicking should reveal a package with the correct structure for an objective-C framework.

UNIVERSAL SUPPORT

Similarly to the Dynamic Framework, you must use a script to generate the target for both platforms and combine their binaries into a single multi-slice binary. This script differs somewhat from the dynamic framework, but only slightly. Since we're essentially merging files that a script rather than Xcode generated, we'll need to refer to them using an environment variable named "STATIC LIB." This script is executed each time you wish to make a fat binary, similar to the dynamic framework. There are many ways to decouple it from our usual build process to allow us more development freedom.

```
set +u

if [[ $SCRIPT_RUNNING ]]
then
exit 0
fi
set -u
export SCRIPT_RUNNING=1

# Environment Variables
FRAMEWORK_NAME=${PROJECT_NAME}
STATIC_LIB="lib${PROJECT_NAME}.a" # Product of other
target
FRAMEWORK_LOCATION="${BUILT_PRODUCTS_DIR}/${FRAMEWORK_
NAME}.framework" # Product of script in other target
build phase
OUTPUT_DIR=${PROJECT_DIR}/Release

# Encapsulate the Xcode Build Process
function build_static_library_and_framework {

    xcrun xcodebuild -project "${PROJECT_FILE_PATH}" \
    -target "${PROJECT_NAME}" \
    -configuration "${CONFIGURATION}" \
```

```
    -sdk "${1}" \
    ONLY_ACTIVE_ARCH=NO \
    BUILD_DIR="${BUILD_DIR}" \
    OBJROOT="${OBJROOT}" \
    BUILD_ROOT="${BUILD_ROOT}" \
    SYMROOT="${SYMROOT}" $ACTION

}

# Encapsulate Lipo
function merge_binaries {

    xcrun lipo -create "${1}" "${2}" -output "${3}"

}

# 1 - Get SDK to determine the platform (iphoneos or
iphonesimulator)
if [[ "$SDK_NAME" =~ ([A-Za-z]+) ]]; then
SDK_PLATFORM=${BASH_REMATCH[1]}
else
echo "Could not find platform name from SDK_NAME:
$SDK_NAME"
exit 1
fi

# 2 - Get the Opposite Platform (iphonesimulator -->
iphoneos, iphoneos --> iphonesimulator)
if [ "$SDK_PLATFORM" == "iphoneos" ]; then
OTHER_PLATFORM=iphonesimulator
else
OTHER_PLATFORM=iphoneos
fi

# 3 - Get the build directories
CURRENT_DIR=${BUILD_DIR}/${CONFIGURATION}-${SDK_
PLATFORM}
OTHER_DIR=${BUILD_DIR}/${CONFIGURATION}-${OTHER_
PLATFORM}

# 4 - Build Opposite Platform
build_static_library_and_framework "${OTHER_PLATFORM}"
```

```
# 5 – Copy the Framework Structure
rm -rf "${OUTPUT_DIR}"
mkdir -p "${OUTPUT_DIR}"
cp -R "${BUILD_DIR}/${CONFIGURATION}-${SDK_
PLATFORM}/${PROJECT_NAME}.framework" "${OUTPUT_
DIR}/${PROJECT_NAME}.framework"

# 6 - Merge Into /Release
merge_binaries "${CURRENT_DIR}/${STATIC_LIB}"
"${OTHER_DIR}/${STATIC_LIB}" "${OUTPUT_DIR}/${PROJECT_
NAME}.framework/${PROJECT_NAME}"
```

After deciding how to include this script, rebuild your target for "Generic iOS Device," and we'll discover our static framework in the/Release directory of our project folder.

UTILIZING OUR STATIC FRAMEWORK

Utilizing our static library in Xcode is equivalent to utilizing any of Apple's dynamically linked-platform frameworks. We do not need to mention ExampleKit.framework as an embedded binary; just add it to our App target's "Linked Libraries and Frameworks" target and verify that our framework bundle resides inside the app project folder. We may get a finished version of the static ExampleKit framework and a properly connected application.

RECOMMENDATIONS

Developing a universal structure is just half of the fight. Suppose we want to provide our framework for usage by more than a few individuals with particular use cases in mind. In that case, there are a few things we should do to ensure that it will operate in a completely foreign codebase regardless of the circumstances.

Creating a decent framework cannot be reduced to a collection of rules; it requires a whole shift in perspective. We must consider that our user is an engineer who may attempt to accomplish various things with our framework. They may get access to secret implementations, subclass things they shouldn't, and put our custom data structures in collections in unanticipated ways. They may create terrible code, retain items in memory for too long, or apply our programming patterns poorly. Our objective is to generate as many potential situations as possible and prepare for them in advance.

Nevertheless, there are a few solutions to a variety of typical issues.

COMPILING AND CONSTRUCTING THE FRAMEWORK

To properly construct our framework for simulators and Real devices such as iPhones, we must first have appropriate architectural settings.

Open our framework's.xcodeproj file, navigate the build settings tab, and search for the Valid Architectures setting. We will notice that the debug and release options already contain the values armv7, arm7s, and arm64; if not, add them; compatibility with real devices requires these architectures.

We may add the Any Simulator SDK option for Simulators in Debug and Release modes. Add x86 64 and other architectures to this setting.

UPLOADING AN APPLICATION'S FRAMEWORK TO THE APP STORE

Ensure the following before creating a successful archive for our software and sending it to the App Store:

- If we want to activate bitcode in our framework, ensure that the bitcode settings have been added to the framework. Go to the framework project settings and, for each target, change the Enable Bitcode option to YES for both Debug and Release modes.

  Search for bitcode configurations. Add -fembed-bitcode to both the Debug and Release modes, or add -fembed-bitcode-marker to the Debug mode and -fembed-bitcode to the Release mode.

  Add BITCODE GENERATION MODE to the User Defined settings section, and then add bit code for both Debug and Release modes, or add a marker in Debug and bitcode in Release mode.

- Open our App project, navigate to the target App settings, and then to the Build Phases page, where we may add the following Run Script to eliminate incompatible architecture before uploading the build.

Mention the input file path as our.framework file path, such as

```
$(SRCROOT)/Frameworks/AppColors/AppColors.framework
```

MEMORY MANAGEMENT IN OBJECTIVE-C

Memory management is a necessary procedure in every programming language. It is the process by which objects' memory is allocated when needed and deallocated when they are no longer needed.

Object memory management is a performance issue; if an application does not release unused objects, its memory footprint expands and performance decreases.

Objective-C memory management approaches may be divided into two categories.

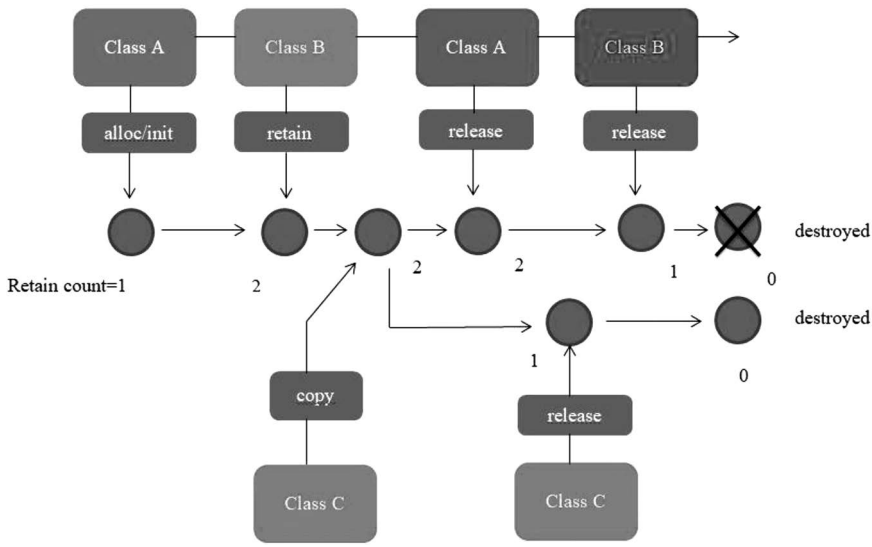
- "Manual Retain-Release" or MRR
- "Automatic Reference Counting" or ARC

"MANUAL RETAIN-RELEASE" OR MRR

In MRR, we manage memory explicitly by keeping track of the items independently. This is accomplished by using a model known as reference counting, which is provided by the Foundation class NSObject in combination with the runtime environment.

The only difference between MRR and ARC is that we handle the retain and release manually in the former while handled automatically in the latter.

The diagram shown below is an example of how memory management works in Objective-C.



Manual retain-release.

The Class A object's memory life cycle is shown in the diagram above. As you can see, the retain count is shown under the object; when the retain

count reaches 0, the item is liberated, and its memory is reallocated for other objects to utilize.

The alloc/init function in NSObject is used to construct the Class A object. The number of retains is now 1.

Class B now keeps Class A's object, and Class A's object now has a retain count of 2.

The item is then copied by Class C. It is now constructed as a new instance of Class A, with the same instance variables. The retain count is 1 instead of the original object's retain count. The second horizontal line in the diagram represents this.

Class C uses the release method to release the copied object, which causes the keep count to drop to 0 and the item to be destroyed.

The keep count for the original Class A Object is 2, and it must be released twice before it may be destroyed. This is accomplished using Class A and Class B release statements to set the keep count to 1 and 0, respectively. The thing is finally destroyed.

Basic MRR Rules

- We own whatever we make: a method that starts with "alloc," "new," "copy," or "mutableCopy" is used to construct an object.
- We may use retained to acquire ownership of an object: a received object is usually guaranteed to stay valid inside the method in which it is received. That method may also return the object to its invoker securely. Retain is used in two scenarios.
    - We use an accessor method or an init method to gain ownership of an object we wish to save as a property value.
    - To avoid the invalidation of an object as a result of another action.
- When we no longer need something, we must give up ownership of it: a release message or an autorelease message is used to relinquish control of an item. Consequently, relinquishing ownership of an item is referred to as "releasing" an object in Cocoa parlance.
- We must not give up ownership of something we don't own: this is a consequence of the previously mentioned policy norms.

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

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

@implementation SampleClass
- (void)sampleMethod {
   NSLog(@"Hello, Everyone \n");
}

- (void)dealloc  {
  NSLog(@"Object deallocated");
  [super dealloc];
}

@end

int main() {

   /* my first program in the Objective-C */
   SampleClass *sampleClass = [[SampleClass alloc]
init];
   [sampleClass sampleMethod];

   NSLog(@"Retain Count after the initial allocation:
%d",
   [sampleClass retainCount]);
   [sampleClass retain];

   NSLog(@"Retain Count after the retain: %d",
[sampleClass retainCount]);
   [sampleClass release];
   NSLog(@"Retain Count after the  release: %d",
[sampleClass retainCount]);
   [sampleClass release];
   NSLog(@"SampleClass dealloc will call before
this");

   // Should set object to nil
   sampleClass = nil;
   return 0;
}
```

"AUTOMATIC REFERENCE COUNTING" OR ARC

Automatic Reference Counting, or ARC, employs the same reference counting approach as MRR, but it inserts the proper memory management method calls for us at build time. For new projects, we are highly recommended to adopt ARC. If we utilize ARC, we usually don't need to know about the underlying implementation detailed in this chapter, while it could be helpful in some instances.

We do not need to introduce release and retain methods in ARC since the compiler will take care of it. Objective-C core methodology hasn't changed. Internally, it makes it simpler for the developer to write without worrying about the retain and release actions, reducing the amount of code written and the risk of memory leaks.

Another notion called garbage collection is utilized in Mac OS X alongside MRR, but it hasn't been acknowledged since its deprecation in OS X Mountain Lion. Furthermore, trash collection was never available for iOS objects. Garbage collection is also not used on OS X while using ARC.

Here's an example of an ARC. Because the online compiler does not support ARC, this will not work.

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

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

@implementation SampleClass
- (void)sampleMethod {
   NSLog(@"Hello, everyone \n");
}

- (void)dealloc  {
  NSLog(@"Object deallocated");
}

@end

int main() {
   /* my first program in the Objective-C */
   @autoreleasepool {
      SampleClass *sampleClass = [[SampleClass alloc]
init];
```

```
        [sampleClass sampleMethod];
        sampleClass = nil;
    }
    return 0;
}
```

Effective Procedures Prevent Memory-Related Issues

There are two primary types of problems caused by improper memory management:

- Releasing or erasing data that is currently in use.
- This often results in the program crashing or, in the worst case, damaged user data.
- Memory leaks result from the failure to release inactive data.
- A memory leak occurs when allocated memory is not released, despite never being utilized again. Our program uses ever-increasing quantities of memory due to memory leaks, which may result in poor system performance or the termination of our application.

DEBUG MEMORY ISSUES USING ANALYSIS TOOLS

We may use the Clang Static Analyzer included in Xcode to find flaws with your code at compilation time.

If memory management difficulties do emerge, there are additional tools and approaches we may use to identify and diagnose the problems.

Many of the tools and approaches, including the usage of NSZombie to aid detect over-released objects, are explained in Technical Note TN2239, iOS Debugging Magic.

Instruments may be used to monitor reference counting events and detect memory leaks.

THE GOAL OF MEMORY MANAGEMENT

The objective of any memory management system is to lower a program's memory footprint. Controlling the lifespan of objects produced inside the application accomplishes this. According to memory management best practices, objects should only live for as long as necessary and not a millisecond longer.

iOS and OS X Apps have developed an object ownership model. An internal reference-counting mechanism keeps track of how many owners

each object has. When an item is claimed as one's own, the reference count increases by one. The opposite also holds true. After the item is released (when it is no longer needed), its reference count is decreased. As long as the reference count is more than zero, the existence of an object is assured. However, once the count approaches zero, the operating system is permitted to delete it.

Historically, developers manually managed an object's reference count using specific memory management methods supplied by the NSObject protocol. The term for this is Manual Retain Release (MRR). Automatic Reference Counting (ARC) is implemented in Xcode 4.2 to insert these method calls automatically.

Avoid Crashing

Manually regulating object ownership may be difficult and time-consuming, mainly because a programmer must remember to surrender possession of the object when using it. This is a routine that is required but not always practicable. This implies that every alloc, retain, and copy call must be paired with a release or autorelease on the same object.

A memory leak or a dangling pointer will happen if these calls are not balanced. If an object is not released, its underlying memory will never be removed, resulting in a memory leak. Minimal leakage will have no discernible impact on your software. However, if enough memory is used, the software will ultimately crash! In contrast, if we repeatedly attempt to release an object, we will have what is known as a dangling pointer. When we attempt to access the hanging pointer, you will be requesting an invalid memory location, and your software will likely fail once again.

Strong vs Weak

Both strong and weak references exist between things. A strong reference implies ownership; a strong property increments the object's reference count. The referent object is the owner of the referred item. A weak reference means that the object referred to does not possess the referenced item. Weak references do not increase the object's reference count. The lifespan of an item is defined by the number of strong references to it. An item is not released as long as it is strongly referenced. The weak identifies a reference that does not maintain the life of the referred object. If there are no strong references to an object, the weak reference is null.

Strong is the default attribute for object types in ARC. An item is considered "alive" as long as a strong pointer points.

Atomic and Nonatomic

Atomicity relates to the behavior of the class or object in a multithreaded environment. Atomic accessors guarantee that the value is entirely set or retrieved in the thread that is accessing the variable.

In setting or retrieving a specific property, an atomic accessor is considered thread-safe.

Atomic accessors do not guarantee thread safety for the whole class. They merely assure that the property is not accessed during the set.

Making accessors atomic reduces speed since the compiler must run more instructions when accessing our values.

Make them atomic when threads are possible.

If we know they are not accessed in a threaded environment, make them non-atomic. Therefore, non-atomic is much quicker than atomic.

Nonatomic is used to indicate that the referenced object is not thread-safe. This indicates that the item cannot handle several requests simultaneously. Defines a reference that does not maintain the object's existence. If there are no strong references to an object, the weak reference is null.

DESIGN PATTERNS IN iOS

iOS design patterns aid in providing reusable solutions to common difficulties such as creating complicated code structures and managing code when building Apps. In a nutshell, it's a template for reusing and understanding code.

There are several design patterns available for creating seamless iOS applications. We may know how to construct an iOS App, but to design iOS Apps successfully and efficiently, we must first grasp iOS design patterns. Although design patterns may not be flashy, and most developers may not pay much attention to them, they are pretty crucial. Design patterns are reusable solutions to typical software development challenges. A design is simply a template that assists developers in writing code that is simple to comprehend and reusable. In addition, the use of design patterns enables the developer to modify or replace code pieces with relative simplicity and speed.

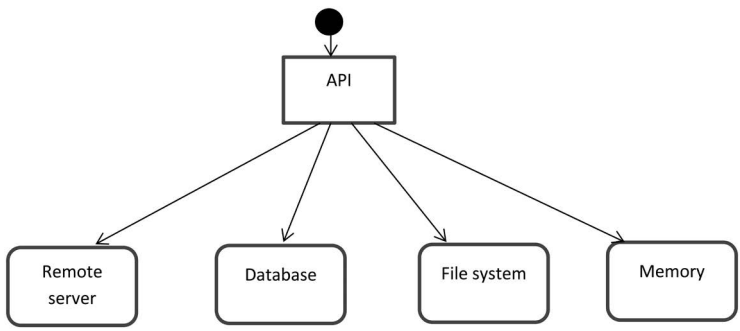
As a result of how Cocoa has been constructed, Cocoa developers will discover that they are already using iOS patterns. These design patterns are in place to aid developers in adhering to best practices to expedite and simplify the development process.

Design patterns are reusable solutions to common software design challenges. They are templates meant to assist us in writing code that is simple to comprehend and reuse. The following are the most frequent Cocoa design patterns:

- Creational: Singleton
- Structural: Decorator, Adapter, and Facade
- Behavioral: Observer and Memento

FAÇADE

The Facade design pattern creates a unified user interface for a complicated subsystem. Instead of providing a collection of classes and APIs to the user, we just offer one basic unified API.



Façade design pattern.

This design helps work with many classes, especially if they are challenging to use or comprehend. If the classes beneath the facade are expected to change, the facade class can keep the same API while things change behind the scenes. For example, if you decide to update our backend service, we will only need to alter the code that utilizes our API, not the code within our Facade.

When to Use the Facade Pattern?

- During the development of a complicated subsystem, design patterns provide a straightforward and consistent interface.
- A Facade design pattern is used to split a subsystem into many levels.

An Illustration of Facade Design Pattern

The computer startup is the best illustration of a Facade design pattern. Many components, such as RAM, hard disc, and motherboard, are required for the computer to boot. A computer will present all capabilities via a single interface to simplify the process.

The Facade design pattern accomplishes the same goal by concealing the system's complexity and facilitating the client's access to the system.

DECORATOR

The Decorator design dynamically adds behaviors and responsibilities to an object without altering its code. It is an alternative to subclassing in which the functionality of a class is altered by wrapping it in another object.

Category and Delegation are two typical implementations of this pattern in Objective-C. Extensions and Delegation are two extremely typical implementations of this paradigm in Swift.

This is used to separate implementation-specific behavior from the generic class. Many iOS UI components, such as UIScrollView, employ delegates to control their behavior. As this is an application-specific job, the UIScrollView class is unaware of the scrolled content. To inform the application of scrolling events, the UIScrollViewDelegate is used. The application may implement the delegate and intercept the scrolling events given by UIScrollView.

When Should We Use a Decorator Pattern?

When an expansion is required, the Decorator design pattern is used. For instance, the Windows operating system features need "optional" components such as the title bar, status bar, and scrollbars.

Objects also implement the "decoration" extension of this design style. These objects have a similar interface, characteristics, superclass, and intermediate superclasses.

Example of Decorator Style Design

The Decorator dynamically assigns additional object responsibilities. The decorations put on pine or fir trees are known as decorators. Decorate a tree with lights, garland, candy canes, glass ornaments, etc.

The decorations have little influence on the overall aesthetic of the tree, which remains recognized as a Christmas tree regardless of its presence. The installation of lights, which can "light up" a Christmas tree, may display additional functionality.

MEMENTO

Memento Pattern keeps our information someplace. This externalized state is later restored without breaking encapsulation; private data stays private. Archiving, Serialization, and State Restoration are examples of Memento pattern implementations.

ADAPTER

The Adapter design pattern transforms a class's interface into another interface that customers anticipate. The adapter makes it possible for classes with conflicting interfaces to collaborate. It decouples the client from the targeted object's class. Apple employs protocols to do its tasks. Protocols like UITableViewDelegate, UIScrollViewDelegate, NSCoding, and NSCopying may be familiar. Using the NSCopying protocol as an example, any class may implement a standard copy function.

When to Use an Adapter?

- When a third-party class is required, its interface does not match the actual application's code.
- In the lack of specialized functionality and limited extensibility, an Adapter is used with existing subclasses.

Illustration of Adapter Pattern

Suppose you have a Bird class with the fly() and create sound methods (). Additionally, the ToyDuck class has a squeak() function. Now that you're short on ToyDuck items, you want to replace them with Bird items. Birds possess comparable skills, but their interface is distinct; thus, we cannot use them directly. Consequently, we will use the Adapter pattern. A bird would be our client, whereas ToyDuck would be our customer.

OBSERVER

The Observer design pattern establishes a one-to-many dependence between objects such that when the state of one objects changes, its dependents are immediately alerted and changed. The Observer pattern is simply a publish-and-subscribe paradigm with loose coupling between the topic and its observers. There may be communication between the

observer and the observed item without each party knowing a great deal about the other. The observer pattern is implemented in Cocoa in two ways: Notifications and Key-Value Observing (KVO). Delegate: This is used to prevent implementation-specific behavior from being included in the generic class. Many iOS UI components, such as UIScrollView, employ delegates to control their behavior. As this is an application-specific job, the UIScrollView class is unaware of the scrolled content. To inform the application of scrolling events, the UIScrollViewDelegate is used. The application may implement the delegate and intercept the scrolling events given by UIScrollView.

When Should We Use a Decorator Pattern?

When an expansion is required, the Decorator design pattern is used. For instance, the Windows operating system features need "optional" components such as the title bar, status bar, and scrollbars.

Objects also implement the "decoration" extension of this design style. These objects have a similar interface, characteristics, superclass, and intermediate superclasses.

Example of Decorator Style Design

The Decorator dynamically assigns additional object responsibilities. The decorations put on pine or fir trees are known as decorators. Decorate a tree with lights, garland, candy canes, glass ornaments, etc.

The decorations have little influence on the overall aesthetic of the tree, which remains recognized as a Christmas tree regardless of its presence. The installation of lights, which can "light up" a Christmas tree, may display additional functionality.

STRATEGY

Strategy pattern permits the modification of an algorithm's behavior during runtime. We may build a family of algorithms using interfaces and encapsulate and make them interchangeable, enabling us to choose which algorithm to execute at runtime.

FACTORY

Factory method design makes adding or deleting new kinds from the codebase easier. We just need a new class and a new factory to create a new type.

COMMAND

The Command design pattern encapsulates a request as an object, allowing you to parameterize clients, queue or log requests, and offer undoable activities. The request object connects one or more operations on a single receiver. The Command pattern distinguishes between objects that make a request and those that receive and execute that request. For example, Mechanism of targeted action.

COMPOSITE

The Composite design pattern assembles linked elements into tree structures to describe part-whole hierarchies. It allows clients to handle individual objects and object combinations consistently. It is a component of the Model-View-Controller aggregation pattern. For example, View Hierarchy.

ITERATOR

The Iterator design pattern enables sequential access to the components of a collection or other aggregate object without disclosing its underlying representation. The Iterator pattern moves the accessing and traversal of a collection's items from the collection itself to an iterator object. Iterator provides an interface for accessing collection items and maintains the current element's position. Different iterators may implement distinct traversal strategies. For example, Enumerators.

MEDIATOR

The Mediator design pattern specifies an item that encapsulates how a collection of things communicate. The mediator encourages loose coupling by preventing objects from explicitly referencing one another, enabling us to modify their interaction independently. This increases the reusability of some items. In this paradigm, a "mediator object" centralizes complicated communication and control logic across system components. These objects inform the mediator object when their status changes and reply to requests made by the mediator object. Instances include Controller Classes in the AppKit Framework and View Controllers in UIKit.

SINGLETON

The Singleton design pattern assures that a class has only one instance and gives a global point of access to that instance. The class maintains track of its one instance and prevents further instances from being generated.

Singleton classes should be used when it makes sense for a single object to offer access to a global resource. Typically, it employs lazy loading to generate the instance only when required for the first time.

When Should We Use the Singleton Design Pattern?

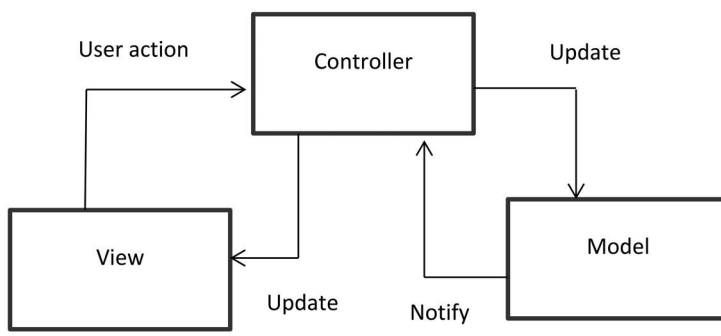
If we are needed to manage resources, you should implement this design technique. We must replace a Singleton with a connection pooling HTTP client that is more efficient and resource-friendly.

Illustration of Singleton Pattern

The Singleton function is indicated when just one instance of a class or a single object copy is required. Global access is allowed for a single instance. It employs a lazy loading strategy to create a single instance on the first attempt.

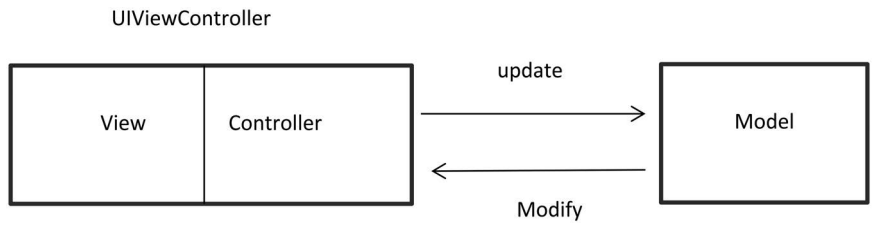
MVC

- Models: These are in charge of the domain data or a data access layer that manipulates the data; for example, consider the "Person" or "PersonDataProvider" classes.
- Views: Views are in charge of the presentation layer (GUI) in the iOS environment; consider everything beginning with the "UI" prefix.
- Controller/Presenter/ViewModel: The glue or mediator between the Model and the View, in general, responsible for adjusting the Model by reacting to user actions on the View and updating the View with Model changes.



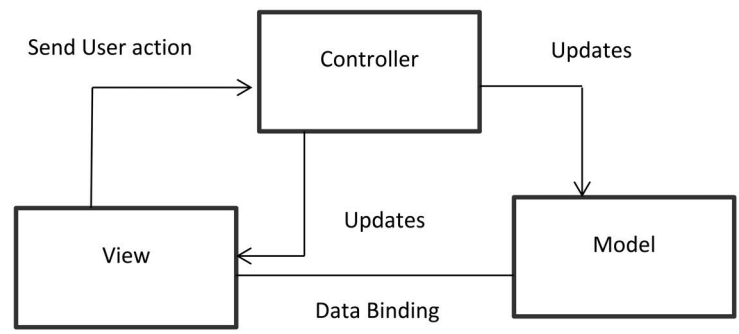
MVC model.

The ViewController is the owner of the Model and contains the View. The issue is that we used to write both the controller and view code in the ViewController. It complicates the ViewController. That's why it's known as a Massive View Controller. When developing a ViewController test, you must mimic the view and its life cycle. However, opinions are tough to ridicule. And we don't want to fake the view if we're simply testing the controller logic. All of these factors contribute to the difficulty of writing examinations.



Viewcontroller of MVC.

MVP



MVP model.

Model View Presenter comprises three elements: the Presenter (UIKit-independent mediator), the Passive View (UIView and/or UIViewController), and the Model. This pattern identifies Views as the receivers of UI events, calling the required Presenter. In reality, it is the responsibility of the Presenter to update the View with the new data supplied by the Model. View has a looser connection to the model. Responsible for linking the Model to the View is the Presenter. Interface-based

interaction with the view facilitates unit testing. Typically, View to Presenter Means one-to-one mapping. There may be several speakers for intricate perspectives.

MVVM

In MVVM, the View solely contains visual aspects such as layout, animation, and initializing UI components. The ViewModel is a specific layer that exists between the View and the Model. ViewModel is the authoritative representation of View. In other words, the ViewModel offers a collection of interfaces, each representing a UI component in the View. We utilize the "binding" approach to link UI components to ViewModel interfaces.

Specifically, for MVVM in iOS programming, the View is represented by UIView/UIViewController. We only do:

- Initiate, layout, and display UI components.
- Connect UI elements to the ViewModel.

In contrast, we perform the following in the ViewModel:

- Create controller logic, including pagination and error handling.
- Write presentational logic and give View interfaces.

Feature Assessment

The MVVM View has more responsibilities than the MVP View, even though this is not immediately apparent in our little example. Because the first one changes its state from the View Model by establishing bindings, the second one transmits all events to the Presenter without updating itself.

Testability – the View Model is ignorant of the View, simply allowing us to test it. The View is checked, but as it is dependent on UIKit, we may wish to avoid it.

Easy to use – it has the same amount of code as MVP in our example, but in the actual App, where we would have to send all events from the View to the Presenter and manually update the View, MVVM would be far slimmer if bindings are utilized.

VIPER

Difficulties with MVVM:

It is compounded on iOS by the absence of bindings and the inclination to continue transferring too many tasks from a view controller class to a view model class.

Without good, reusable, tested, and single-responsibility components, the use of any MVVM solution is limited.

Simply stating "we're using MVVM" or having a rudimentary implementation of MVVM in a project without analyzing how it improves our code reuse, efficiency, and adherence to the single responsibility principle can mislead us on how beneficial or maintainable the design is.

VIPER applies Clean Architecture to iOS applications. VIPER in Objective-C is an acronym that stands for View, Interactor, Presenter, Entity, and Routing. Clean Architecture separates the logical framework of an application into different levels of responsibility. This makes it easy to separate dependencies (such as our database) and test interactions at layer boundaries.

The main VIPER components are:

- View: It shows what the Presenter instructs it to and communicates user input to the Presenter.
- Interactor: It includes business logic given by a use case.
- Presenter: It contains view logic for preparing material for display (as provided by the Interactor) and responding to user inputs (by requesting new data from the Interactor).
- Entity: It stores fundamental model items used by the Interactor.
- Routing: It contains navigation logic detailing the sequence in which screens are shown.

WHAT ARE THE ADVANTAGES OF EMPLOYING iOS DESIGN PATTERNS?

The design pattern improves the definition of interfaces and is device-specific. However, it is entirely up to us to use the code and add it to a particular template. The table below summarizes the advantages of employing iOS design patterns.

Benefit	Explanation
Bring Tested Solutions	Design patterns assist you in resolving software development challenges and guiding us through all of the phases.
Code Unification	Design patterns also help to tackle the problem by providing standard solutions that have been tested for specific faults and restrictions. It will help us identify errors in the App's architectural design when organizing and developing.
Common Vocabulary	Other developers can readily grasp the solutions you have created to meet the difficulty by just expressing the name of a given design pattern.

Prepared to Develop iOS Applications Using iOS Design Patterns

This tutorial provides top iOS design patterns that are simply used in iOS applications. We discovered the definition, examples, and justifications for using iOS design patterns.

Using the mentioned iOS design patterns may significantly enhance our App development.

Consult our seasoned App development team if we want to design iOS mobile applications. As the leading iOS mobile App development firm, we can assist in creating a fully working iOS application utilizing the most recent iOS design principles.

In this chapter, we covered code management with its relevant examples.

BIBLIOGRAPHY

1. Obj-C Memory Management.
2. Advanced Memory Management Programming Guide.
3. iOS: Design Patterns: Chetan Aggarwal.
4. Nine Best iOS Design Patterns to Develop Powerful iPhone Apps.
5. Obj-C Memory Management – https://www.tutorialspoint.com/objective_c/objective_c_memory_management.htm, accessed on May 13, 2022.
6. Memory management – https://livebook.manning.com/book/objective-c-fundamentals/chapter-9/, accessed on May 13, 2022.
7. Programming in Objective-C: Creating Your First Program – https://www.informit.com/articles/article.aspx?p=2159356, accessed on May 13, 2022.
8. Creating Your First Objective-C Application – https://andybargh.com/create-your-first-objective-c-application/, accessed on May 13, 2022.
9. Creating your custom Objective-C Framework for iOS Apps – https://prathma.medium.com/creating-your-custom-objective-c-framework-for-ios-apps-5d5ccf95c6c7, accessed on May 13, 2022.
10. About Memory Management – https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/MemoryMgmt.html, accessed on May 13, 2022.
11. Memory Management in Objective-C – https://medium.com/@JanlCodes/memory-management-in-objective-c-7acc36d20caf, accessed on May 13, 2022.
12. Best iOS Design Patterns to Develop Powerful iPhone Apps – https://www.spaceotechnologies.com/blog/ios-design-patterns, accessed on May 13, 2022.

CHAPTER 6

Code Optimization

IN THIS CHAPTER

- Optimization Tips
- Writing Secure Code
- Best Coding Practices
- Security and Hardening Ideas

The previous chapter covered frameworks, memory management, and system design in Objective-C. In this, we will discuss code optimization.

OBJECTIVE-C CODE OPTIMIZATION AT COMPILE TIME

Code optimization is developing or rewriting code so that a program utilizes the least amount of memory or disk space, requires the least CPU time or network bandwidth, or makes the most effective use of additional cores.

Code compilation is one of the necessary procedures (together with linking, bundling the resources, code signing, and so on).

Our source code was initially created in a high-level programming language, such as Objective-C, Swift, or C++. It is the compiler's responsibility to convert our high-level source code into low-level machine code to generate an executable program. There are other sorts of compilers, but in the iOS/macOS ecosystem, we only deal with frontend and backend compilers. The frontend translates our source code into an intermediate language that the backend can comprehend. This intermediate representation

DOI: 10.1201/9781003357506-6

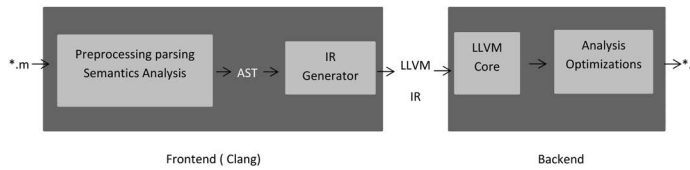
is converted into machine code by the backend, taking into account the individual operating system and processor architecture. This is the core of these two entities.

Different pipelines exist for Swift and Objective-C, although LLVM supports both.

LLVM is a large project that consists of LLVM Core as the backend, and Clang as the "LLVM native" frontend for the C language family (C/Objective-C/C++/Objective-C ++), and an interaction protocol between the frontend and backend. The primary component of this protocol is LLVM IR (Intermediate Representation) – the intermediate language that can be understood by both the front and back ends. It is irrelevant to LLVM Core whatever front-end generated the source code's IR-code. The Core simply manipulates this representation to build an executable or dynamic library.

OBJECTIVE-C PIPELINE

So the pipeline for good old Objective-C code looks like this:



Objective-C pipeline.

As you can see, the first conversion Clang does on our code is the development of an AST' (Abstract Syntax Tree) – a representation in which all of the functions, operators, variables, declarations, etc. are nodes of a massive semantic tree. Here's an AST example:

```
$ cat test1.cc
 int f(int y) {
  int result = (y / 42);
  return result;
 }
```

which then turns into:

```
$ clang -Xclang -ast-dump -fsyntax-only test1.cc
 TranslationUnitDecl 0x5aea0d0 &lt;&lt;invalid
sloc&gt;&gt;
```

```
 ... cutting out internal declarations of clang ...
 '-FunctionDecl 0x5aeab50 &lt;test.cc:1:1,
line:4:1&gt; f 'int (int)'
  |-ParmVarDecl 0x5aeaa90 &lt;line:1:7, col:11&gt; x
'int'
  '-CompoundStmt 0x5aead88 &lt;col:14, line:4:1&gt;
   |-DeclStmt 0x5aead10 &lt;line:2:3, col:24&gt;
   | '-VarDecl 0x5aeac10 &lt;col:3, col:23&gt; result
'int'
   |  '-ParenExpr 0x5aeacf0 &lt;col:16, col:23&gt;
'int'
   |   '-BinaryOperator 0x5aeacc8 &lt;col:17,
col:21&gt; 'int' '/'
   |    |-ImplicitCastExpr 0x5aeacb0 &lt;col:17&gt;
'int' &lt;LValueToRValue&gt;
   |    | '-DeclRefExpr 0x5aeac68 &lt;col:17&gt; 'int'
lvalue ParmVar 0x5aeaa90 'x' 'int'
   |    '-IntegerLiteral 0x5aeac90 &lt;col:21&gt;
'int' 42
   '-ReturnStmt 0x5aead68 &lt;line:3:3, col:10&gt;
    '-ImplicitCastExpr 0x5aead50 &lt;col:10&gt; 'int'
&lt;LValueToRValue&gt;
     '-DeclRefExpr 0x5aead28 &lt;col:10&gt; 'int'
lvalue Var 0x5aeac10 'result' 'int'
```

The AST is then translated into LLVM IR, which is lower level (less human-readable), but some parts of the source code may still be understood (a comprehensive description of the language can find here):

```
int main()
{
 return 0;
}
```

which then turns into

```
define i32 @main() #0 {
    %1 = alloca i32, align 4
    store i32 0, i32* %1
    ret i32 0
   }
```

Clang passes LLVM IR to LLVM Core, optimizing (if appropriate) and converting the code to target-specific machine code. Consequently, we have a collection of object files (*.o) that are subsequently linked and combined into an executable or dynamic library. This final output is often referred to as a "a.out," ".dylib," or ".so" file.

As was previously evident, LLVM Core is where code optimization occurs, and Intermediate Representation is the source of these optimizations.

Here are the LLVM optimization levels and their short descriptions:

- None [-O0]: The compiler does not optimize source code. With this option, the compiler aims to decrease the cost of compilation and deliver anticipated results from debugging. Statements are independent: if you pause the program with a breakpoint between statements, you may add a new value to any variable or move the program counter to any other statement in the function and get the expected results from the source code. Use this option during development when we intend to resolve logic issues and need a quick compilation time. Do not utilize this shipping option for our executable.
- Fast [-O, O1]: The compiler performs simple optimizations to improve code performance while reducing the effect on compile time. Additionally, this option consumes more RAM during compilation.
- Faster [-O2]: The compiler executes almost all available optimizations that do not need a trade-off between space and time. This option prevents the compiler from making loop unrolling and function inlining. This option boosts both compilation time and produced code performance.
- Fastest [-O3]: The compiler does all optimizations to increase the resulting code's performance. This option may cause the resulting code to be larger since the compiler will aggressively inline routines. (This choice is often not advised.)
- Fastest, Smallest [-Os]: The compiler does all optimizations that do not normally increase the size of the source code. This is the preferable method for shipping code since it reduces the memory footprint of the executable.

- Quickest and Aggressive Optimisation [-Ofast]: This option allows "Fastest" and aggressive optimizations that may violate stringent standards compliance but should function well with well-behaved code.

SECURE CODE

- **Incorrect platform usage:** To save sensitive data, use adequate keychain security.
  - Keychain safety: Make certain that the program deletes the user's data when the user logs out.

Keychain data linked with the program is not cleared off by default following uninstalling. If a user reinstalls the software and attempts to login with a different user, it may unintentionally provide access to the prior user's account. To avoid this, ensure that the related application's keychain data is checked and deleted when it is launched for the first time after reinstallation. The following code may use to do the wiping procedure:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)
launchOptions
{
    //Clear keychain on the first run in case of
reinstallation
    if (![[NSUserDefaults standardUserDefaults]
objectForKey:@"FirstRun"]) {
        // Delete the values from the keychain
here
        [[NSUserDefaults standardUserDefaults]
setValue:@"1strun" forKey:@"FirstRun"];
        [[NSUserDefaults standardUserDefaults]
synchronize];
    }
}
```

According to the kSecAttrAccessible property, iOS provides the following restricted protection class:

```
- kSecAttrAccessibleAlwaysThisDeviceOnly: Keychain
data can always access, regardless of device is
```

```
locked or not. These data won't include in an
iCloud or local backup.
- kSecAttrAccessibleAfterFirstUnlock: Keychain
data can't be accessed after a restart until the
device has been unlocked once by the user.
- kSecAttrAccessibleAfterFirstUnlockThis
DeviceOnly: Keychain data can't be accessed after
a restart until the device has been unlocked once
by a user. This attribute items do not move to a
new device. As a result, when restoring from a
backup of a different device, these items will be
missing.
- kSecAttrAccessibleWhenUnlocked: Keychain data
can be accessed only while the user unlocks the
device.
- kSecAttrAccessibleWhenUnlockedThisDeviceOnly:
The data in Keychain item can be accessed only
while the user unlocks the device. The data won't
include in an iCloud or local backup.
- kSecAttrAccessibleWhenPasscodeSetThis
DeviceOnly: Keychain data can be accessed only
when the device is unlocked. This protection class
is only available if a passcode is set on the
device. The data won't include in an iCloud or
local backup.
```

- **Unsafe data storage:** It is not suggested to store sensitive data on the client-side.

  Only the refresh token may be kept locally on the keychain if an application utilizes OAuth for login.

To login while using offline authentication, use Password-Based Key Derivation Function 2.

Security Breach through HTTPS Response Cache

After logging out, it is best to delete any cached replies.

```
[[NSURLCache sharedURLCache] removeAllCached
Responses];
```

Implement Empemeral configuration property of the URLSession instead of the cookie, which store session-related data in RAM instead of the local cache ..ephmeral

The cache can also disable by setting cache policy to .notAllowed. It prevents storing cache in memory or disk.

RESUME BACKGROUND DISCLOSURE OF SCREENSHOT DATA

```
@property (UIImageView *)backgroundImage;

- (void)applicationDidEnterBackground:(UIApplication
*)application {
    UIImageView *myBanner = [[UIImageView alloc]
initWithImage:@"overlayImages1.jpg"];
    self.backgroundImage = myBanner;
    [self.window addSubview:myBanner];
}
```

- **Unsafe communication:** It is not suggested to utilize insecure protocols such as HTTP, FTP, and so on.

Sending sensitive data across other channels should be avoided (e.g., SMS, MMS, or notifications).

All communication should take place over a secure channel that supports the following:

SSL versions of protocols are not supported.

Ciphers with less than 128 bits are not supported.

Disable the cipher suites NULL, MD5, SHA1, MD4, RC4, and CBC.

SSL certificates must be valid and issued by a CA.

App Transport Security is enabled by default in iOS 9 and later (ATS). Correctly implement ATS Exceptions.

Ensure that no self-signed or incorrect certificates are allowed in any NSURL calls.

It is recommended that only TLSv1.2 and TLSv1.3 be supported.

To prevent MiTM attacks, use SSL pinning.

SSL PINNING

This may accomplish by developing it from scratch or by utilizing one of the frameworks listed below:

AFNetworking offers a comprehensive network library solution.

TrustKit is an open-source framework that simplifies the implementation of SSL pinning on both Objective-C and Swift.

Apple's NSURLSession API is used for secure communication.

```
(void)URLSession:(NSURLSession *)session didReceiveCha
llenge:(NSURLAuthenticationChallenge *)challenge
completionHandler:(void (^)
(NSURLSessionAuthChallengeDisposition, NSURLCredential
* _Nullable))completionHandler {

    // Get the remote certificate
    SecTrustRef serverTrust = challenge.
protectionSpace.serverTrust;
    SecCertificateRef certificate = SecTrustGetCertifi
cateAtIndex(serverTrust, 0);

    // Set the SSL policies for domain name check
    NSMutableArray *policies = [NSMutableArray array];
    [policies addObject:(__bridge_transfer id)
SecPolicyCreateSSL(true, (__bridge CFStringRef)
challenge.protectionSpace.host)];
    SecTrustSetPolicies(serverTrust, (__bridge
CFArrayRef)policies);

    // Evaluate the server certificate
    SecTrustResultType result;
    SecTrustEvaluate(serverTrust, &result);
    BOOL certificateIsValid = (result ==
kSecTrustResultUnspecified || result ==
kSecTrustResultProceed);

    // Get the local and remote cert data
    NSData *remoteCertificateData = CFBridgingRelease(
SecCertificateCopyData(certificate));
    NSString *pathToCert = [[NSBundle mainBundle]
pathForResource:@"github.com" ofType:@"cer"];
    NSData *localCertificate = [NSData dataWithContent
sOfFile:pathToCert];

    // pinnning check
    if ([remoteCertificateData
isEqualToData:localCertificate] && certificateIsValid)
{
        NSURLCredential *credential = [NSURLCredential
credentialForTrust:serverTrust];
```

```
        completionHandler(NSURLSessionAuthChallenge
UseCredential, credential);
    } else {
        completionHandler(NSURLSessionAuthChallenge
CancelAuthenticationChallenge, NULL);
    }
}
```

- **Unsafe authentication:** Implement the following session depending on the application's business requirements:
  - We propose terminating the user's session if it is inactive for more than 5 minutes in highly sensitive applications.
  - If the program returns from a background state, users must be re-authenticated.
  - If the user hit the logout button, correctly terminate the user's session on both the client and server sides.

  Because ephemeral sessions do not employ persistent storage, they are the best practice.
- **Inadequate cryptography:** The preferred technique for symmetric encryption is AES, with a key of at least 128 bits and a safe mode.

  Asymmetric encryption should use RSA with 2048 bits or ECC with a safe curve.

Use arc4random() for random number generators instead, as it is an unsafe method in Objective-C.

For safe random number generation, use SecRandomCopyBytes.

Avoid employing the cryptographic methods described below since they are insecure.

- DES, 3DES
- RC2
- RC4
- BLOWFISH
- MD4
- MD5
- SHA1

The algorithms listed below are recommended:

Algorithms for confidentiality: AES-GCM-256 or ChaCha20-Poly1305.

SHA-256, SHA-384, SHA-512, Blake2, and the SHA-3 family are examples of integrity algorithms.

Algorithms for digital signatures: RSA (3072 bits and higher), ECDSA with NIST P-384.

RSA (3072 bits and higher), DH (3072 bits and higher), ECDH with NIST P-384.

- **Insecure authorization:** Applications should avoid utilizing id references with guessable numbers.

Server-side authorization checks should perform.

Enforce authorization checks on all server-side requests.

- **Client code reliability**
- Overflow of Buffers
  - Avoid using unsafe functions like sprintf(), gets(), and getpw ()
  - It is recommended that a limit size be enforced for str buffer gets s(str, sizeof(str));
  - To avoid overflow or underflow attacks, provide the following security flags in your application.
    - While developing Apps, enable Address Space Layout Randomization (ASLR). It is enabled by default in iOS 4.3 and macOS 10.7. Prior to this release, we had to explicitly activate it by including the -pie parameter.
    - The application should build with the Non-Executable (NX) flag set (-allow stack execute). This enables the operating system to identify specific sections of memory as non-executable.
    - To reduce the possibility of corruption, enable stack canaries.
      - The -fstack-protector option allows stack canaries only for functions with buffers larger than 8 bytes (e.g., a string on the stack), and is enabled by default when building for macOS 10.6 and later.
      - The -fstack-protector-all switch allows stack canaries to be enabled for all functions.

  - D_FORTIFY_SOURCE: It adds static and dynamic bounds checking to a number of routines that would otherwise have none (sprintf, vsprintf, snprintf, vsnprintf, memcpy, mempcpy, memmove, memset, strcpy, stpcpy, strncpy, strcat and strncat). Only compile-time inspection is performed if the level is set to 1 (-D_FORTIFY_SOURCE=1). When building for macOS 10.6 and later, Level 1 is enabled by default. Level 2 includes extra run-time verification.
  - MallocCorruptionAbort: An environment variable that instructs 32-bit Apps to abort malloc calls if the heap structure is corrupted. Aborting on heap corruption is enabled by default in 64-bit Apps.

- Control flow expressions like "if," "for," "while," "switch," and "try" should not be too nested.
- In the case of an exception, implement an appropriate error-handling mechanism:

```
void save() {
    try {
        saveDocument();
    } catch (const std::exception& ex) {
        log << "Exception while saving document: "
<< ex.what();
    }
    }
```

- **Tampering with the Code:** The Mobile Security Testing Guide describes the following factors that might make debugging an application difficult:[1]

  ptrace: This is the most popular system call used by Apps to observe and manipulate other processes.

- **Excessive functionality:** Remove any secret backdoor functionality or other internal development security controls that are not meant to be released into production.

Check all security protections are turned on in the production environment.

In the production environment, do not save any sensitive data in the log file.

BEST PRACTICES WITH OBJECTIVE-C CODING CONVENTION

The majority of these standards are intended to align with Apple's documentation and community-accepted best practices. Some are influenced by personal tastes. This section tries to provide a standard method of doing things so that everyone follows the same procedure. It is suggested to do something we do not particularly enjoy in order to be consistent with everyone else.

This section is primarily intended for iOS development, although it also applies to Mac.

Operators

```
NSString *foo = @"bar";
NSInteger answer = 52;
answer += 8;
answer++;
answer = 50 + 2;
```

To be consistent with other operators, the ++, −−, and so on are placed after the variable rather than before it. Unless there is just one operand, operators should always be separated by spaces.

Types

According to Apple's best practices and 64-bit safety, NSInteger and NSUInteger should be used instead of int, long, and so on. For the same reasons, CGFloat is recommended over float. This code is future-proofed for 64-bit systems.

Apple types should be preferred over basic ones. When working with time intervals, for example, use NSTimeInterval rather than double, even if they are equivalent. This is considered recommended practice and results in more readable code.

Methods

```
- (void)someMethod {
    // Do the stuff
}
```

```
- (NSString *)stringByReplacingOccurrencesOfString:(NS
String *)target withString:(NSString *)replacement {
    return nil;
}
```

Always leave a space between the – or + and the return type ((void) in this case). A space should never separate the return type and method name.

A space should never use before or after a colon. There should always be a space between the class and the * if the argument type is a pointer.

Always provide a space between the method's finish and the opening bracket. The initial bracket should never appear on the line after it.

Between methods, there should always be two new lines. This corresponds to several Xcode templates (though they vary somewhat) and improves readability.

Pragma Mark and Implementation Organization

An example of a UIView:

```
#pragma mark - NSObject
- (void)dealloc {
    // Release..
    [super dealloc];
}

#pragma mark - UIView
- (id)layoutSubviews {
    // Stuff..
}

- (void)drawRect:(CGRect)rect {
    // Drawing-code
}
```

Methods should group by inheritance. If some UIResponder methods were utilized in the preceding example, they should be placed between the NSObject and UIView methods because that is where they fall in the inheritance chain.

Control Structures

After the control structure, there should always be a space (i.e., if, else, etc.).

If/Else

```
if (button.enabled) {
    // Stuff..
} else if (otherButton.enabled) {
    // Other stuff..
} else {
    // More stuff..
}
```

Else statements should start on the same line as the previous if statement.

```
// Comment explaining conditional
if (something) {
    // Do-stuff
}

// Comment explaining alternative
else {
    // Do the other stuff
}
```

If we want to include comments around the if and else statements, use the structure shown above.

Switch

```
switch (something.state) {
    case 0: {
        // Something..
        break;
    }

    case 1: {
        // Something..
        break;
    }

    case 2:
    case 3: {
        // Something..
        break;
    }
```

```
    default: {
        // Something..
        break;
    }
}
```

Each case should have brackets surrounding it. If more than one case is used, they should be on distinct lines. The default case should always be the last case and present.

For

```
for (NSInteger x = 0; x < 10; x++) {
    // Do something..
}

for (NSString *key in dictionary) {
    // Do something..
}
```

When iterating with integers, it is preferable to begin at 0 and use rather than begin at 1 and use <=. In general, fast enumeration is desired.

While

```
while (something < somethingElse) {
    // Do-something..
}
```

Import

Always use @class in header files instead of #import whenever feasible because it gives a modest compile-time performance gain.

Because the @class directive reduces the amount of code viewed by the compiler and linker, it is the easiest approach to provide a forward declaration of a class name. Because it is basic, it avoids possible issues that may arise when importing files that import other files. If one class defines a statically typed instance variable of another class and their two interface files import one other, neither class will compile successfully.

Header Prefix

It is preferable to include frameworks that are utilized in the majority of a project with a header prefix. If these frameworks are in the header prefix, they should never import into project source files.

For example, consider the following header prefix:

```
#ifdef __OBJC__
    #import <Foundation/Foundation.h>
    #import <UIKit/UIKit.h>
#endif
```

#import <Foundation/Foundation.h> should never occur in the project outside of the header prefix.

Properties

```
@property (nonatomic, retain) UIColor *topColor;
@property (nonatomic, assign) CGSize shadowOffset;
@property (nonatomic, retain, readonly)
UIActivityIndicatorView *activityIndicator;
@property (nonatomic, assign, getter=isLoading) BOOL
loading;
```

If the attribute is nonatomic, it must come first. The next choice should always keep or assign since omitting it results in a warning. If it is given, readonly should be the next choice. In header files, readwrite should never be provided. Only in class extensions should readwrite be utilized. Last should be the getter or setter. Setter should use sparingly.

Private Methods and Properties

MyShoesTier.h

```
@interface MyShoesTier : NSObject {

    .....
}

@property (noatomic, retain, readonly) MyShoe *shoe;

.....

@end
```

MyShoesTier.m

```
#import "MyShoesTier.h"

@interface MyShoesTier ()
- (void)_crossLace:(MyLace *)firstLace
withLace:(MyLace *)secondLace;
@property (nonatomic, retain, readwrite) MyShoe *shoe;
@property (nonaomic, retain) NSMutableArray *laces;
@end

@implementation MyShoesTier

...

@end
```

Because a named category cannot be used if it adds or alters any properties, private methods should always introduce in a class extension.

Note: The above example demonstrates an authorized usage of a read-write attribute.

Extern, Const, and Static

```
extern NSString *const kMyConstant;
extern NSString *MyExternString;
static NSString *const kMyStaticConstant;
static NSString *staticString;
```

Naming

Everything should prefix with a 2–3 letter prefix in general. Longer prefixes are permissible but not preferred.

If the code is application-specific, it is a good idea to prefix classes with an application's particular application. If we want to use the code in other Apps or open-source, it is a good idea to do something unique to us or our firm for the prefix.

Here are a few examples if our firm is Awesome Buckets and you have an application called Bucket Hunter:

```
ABLoadingView // Simple view that can use in other
applications
```

```
BHAppDelegate // Application specific-code
BHLoadingView // 'ABLoadingView' customized for Bucket
Hunter application
```

Enums

```
enum {
    Foo,
    Bar
};

typedef enum {
    SSLoadingViewStyleLight,
    SSLoadingViewStyleDark
} SSLoadingViewStyle;

typedef enum {
    SSHUDViewStyleLight = 9,
    SSHUDViewStyleDark = 13
} SSHUDViewStyle;
```

HARDENING OF SYSTEMS

Systems hardening refers to a set of tools, approaches, and best practices for reducing vulnerability in technology applications, systems, infrastructure, firmware, and other domains. The purpose of system hardening is to decrease security risk by removing potential attack channels and compressing the attack surface of the system. By deleting unnecessary programs, functions, Apps, ports, permissions, access, and so on, attackers and malware have less possibilities to build a foothold within our IT environment.

System hardening necessitates a systematic strategy to auditing, identifying, closing, and controlling any security vulnerabilities throughout our firm.

Hardening of Systems to Reduce the "Attack Surface"

The "attack surface" is the sum of all potential faults and backdoors in technology that hackers can exploit. These flaws can manifest themselves in a variety of ways, including:

- Passwords, both default and hardcoded.
- Plain text files are used to store passwords and other credentials.

- Vulnerabilities in unpatched software and firmware.
- BIOS, firewalls, ports, servers, switches, routers, or other infrastructure components that are not properly set.
- Network traffic or data at rest that is not encrypted.
- Inadequacy or lack of privileged access controls.

Advantages of System Hardening

Continuous work is required for system hardening; however, the investment will pay off in significant ways across your business via:

- Enhanced system functionality: Because there are fewer applications and functions, there is less danger of operational errors, misconfigurations, incompatibilities, and compromise.
- Significantly better security: A smaller attack surface means less danger of data breaches, illegal access, system hacking, or malware.
- Simplified compliance and audibility: Because there are fewer programs and accounts and a less complicated environment, auditing the environment is typically more visible and accessible.

This chapter covered code optimization, where we discussed system hardening, Secure code and code optimization at compile time.

NOTE

1. Secure Code Wiki

BIBLIOGRAPHY

1. Xcode Build Optimization: A Definitive Guide – https://flexiple.com/ios/xcode-build-optimization-a-definitive-guide/, accessed on May 14, 2022.
2. Compile-time code optimization for Swift and Objective-C – https://dmtopolog.com/code-optimization-for-swift-and-objective-c/, accessed on May 14, 2022.
3. Improving build efficiency with good coding practices – https://developer.apple.com/documentation/xcode/improving-build-efficiency-with-good-coding-practices, accessed on May 14, 2022.
4. Optimizing Objective-C – https://www.informit.com/articles/article.aspx?p=1676715&seqNum=6, accessed on May 14, 2022.

5. How To Boost Xcode's Compile Time and Runtime – https://betterprogramming.pub/improve-xcode-compile-and-run-time-8b8f812c17f8, accessed on May 14, 2022.
6. COD 316 – Creating Secure iOS Code in Objective-C – https://www.securityinnovation.com/course-catalog/creating-secure-ios-code-in-objective-c/, accessed on May 14, 2022.
7. Creating Secure iOS Code in Objective-C – https://niccs.cisa.gov/education-training/catalog/security-innovation/creating-secure-ios-code-objective-c, accessed on May 14, 2022.
8. Objective-C Runtime Security and Obfuscation – https://kolinsturt.github.io/lessons/2013/12/12/secure-coding-in-iOS, accessed on May 14, 2022.
9. Objective-C – https://securecode.wiki/docs/lang/objectivec/, accessed on May 14, 2022.

Appraisal

Objective-C programming language is a general-purpose, object-oriented programming language that extends the C programming language with Smalltalk-style messaging. Before the introduction of Swift, it was the primary programming language used by Apple for the OS X and iOS operating systems and their corresponding application programming interfaces (APIs): Cocoa and Cocoa Touch. In the early 1980s, the programming language Objective-C was created. It was chosen as the primary programming language for NeXT's NeXTSTEP operating system, from which OS X and iOS inherit. Portable Objective-C applications that do not utilize the Cocoa or Cocoa Touch libraries, or use components that may be ported or re-implemented for other platforms, can also be built for any system supported by GNU Compiler Collection (GCC) or Clang.3

Objective-C "implementation" software files typically have.m filename extensions, but Objective-C "header/interface" files have.h filename extensions, the same as C header files. The file extension for Objective-C++ documents is .mm.

Objective-C was built principally by Brad Cox and Tom Love at their firm Stepstone in the early 1980s. Both were exposed to Smalltalk at the Programming Technology Center of ITT Corporation in 1981. The first work on Objective-C dates back to about that time.

In 1986, Cox presented the initial explanation of Objective-C in his book Object-Oriented Programming, An Evolutionary View. Although he was cautious to note that the issue of reusability involves more than simply the language, Objective-C was often compared with feature of other languages. In 1988, the language was licensed, and the NeXTSTEP code library was built.

Apple Computer utilized OpenStep in its new operating system, Mac OS X, after purchasing NeXT in 1996. This includes Objective-C, Project Builder, NeXT's Objective-C-based developer tool, and Interface Builder, its interface design tool (both now merged into one Xcode application).

DOI: 10.1201/9781003357506-7

The majority of Apple's current Cocoa API is built on OpenStep interface objects, and it is the most important Objective-C development environment in use today.

Swift, which Apple launched at WWDC 2014, was described as "Objective-C without the C." The Objective-C programming language has a lengthy history. Although it has languished as a niche language for most of that time, the arrival of the iPhone has rocketed it to popularity: Objective-C was named the winner of the TIOBE Programming Language Award for 2011 in January 2012. This award is given to the language with the highest use growth over the past 12 months; in 2011, Objective-C jumped from eighth to the fifth position on the index.

The origins of Objective-C may be traced to a section of International Telephone and Telegraph (ITT) where Tom Love and Brad Cox sought to increase the productivity of programmers. In 1981, a special edition of Byte magazine presented Smalltalk, a breakthrough new programming language created by Alan Kay and his colleagues at Xerox PARC. Smalltalk introduced a whole new way of conceiving program development, which Kay termed "object-oriented." Instead of writing a program as a set of operations ("procedures") that accept data as inputs, a program would be re-oriented around the data itself, which would be organized into "objects." Smalltalk envisioned programs as collections of objects that would exchange messages, therefore invoking "methods" (perform actions). The invoked methods were chosen "dynamically," that is, while the program was executing. A Smalltalk application may edit itself in real-time in reaction to human input.

Objective-C has been around for 35 years, making it older than many current macOS and iOS software developers. Swift, which debuted in 2014, is swiftly gaining momentum and displacing Objective-C from the market. Observations indicate that about half of the most popular applications in the App Store are created in Swift, which is a significant increase in just 5 years.

The million-dollar issue is whether developers should consider Objective-C for new iOS projects or go for Swift from the beginning. Let's examine the present state of Objective-C.

First, nobody has ever heard any official word from Apple on the deprecation of Objective-C support or even a possible date when it would be discontinued. A large amount of code created in Objective-C is still being maintained, and a large number of popular applications are built in this language. Consider: who would commit to transferring all those

applications to a different language simply for its purpose? Imagine the amount of time and money that such a project would need with no apparent return.

Thus, we can conclude that Objective-C will continue to exist in the foreseeable future. Objective-C programmers may rest confident that their talents will continue to be in demand. With new applications and young developers picking the language to learn, the situation is different.

A COMPARISON OF OBJECTIVE-C AND SWIFT

Let's analyze how Objective-C compares to Swift and what might tip the scales in this matter to determine if it's a smart idea to choose it as the language for our new iOS App project.

Let's explore the differences between Objective-C and Swift in further depth.

- Objective-C's performance is sluggish owing to runtime code compilation. According to an official Apple release, Swift is 2.6 times more efficient than Objective-C.
- Objective-C inherits its code structure from C and, as a result, has a syntax with several special symbols that is reasonably difficult. On the other hand, Swift is similar to plain English, making its learning curve simpler than that of Objective-C.
- The primary difference in code safety is how Objective-C and Swift handle null pointers. A null pointer in Objective-C does not result in an application crash, making it challenging to identify and repair. In Swift, however, null pointers result in easily identifiable crashes. Overall, Objective-C applications may include more hidden problems than Swift applications.
- Objective-C demands the maintenance of two code files: time- and effort. With Swift, we need to maintain a single code file, reducing the time and human labor required.
- Objective-C often requires more lines of code than Swift.
- Objective-C lacks support for dynamic libraries and instead utilizes static ones. They are bulkier and hinder application performance. Swift utilizes dynamic libraries that decrease the size of the whole application and improve its performance and quickness.

Sr. No.	Swift	Objective-C
01.	Swift is a high-level, general-purpose programming language that prioritizes safety and speed.	Objective-C is a general-purpose programming language regarded as a superset of the C programming language. It was created to provide object-oriented features.
02.	Chris Lattner created it in conjunction with Apple's other programmers.	Brad Cox and Tom Love created it at their firm, Stepstone.
03.	Objective-C, Rust, Ruby, and Python affected it.	The language was inspired by C and Smalltalk.
04.	Swift appeared for the first time in 2014.	Objective-C was initially introduced in 1984.
05.	Swift type is static.	The type of Objective-C is dynamic.
06.	Swift is an open-source, Apache-licensed project.	The licensing for Objective-C is General Public License.
07.	It has just classes.	Both Structs and classes are present.
08.	It was built for developing iOS, Mac, Apple TV, and Apple Watch applications.	Objective-C is created with Smalltalk communications capabilities in mind.
09.	Swift polymorphism is not directly present.	In Objective-C, polymorphism exists immediately at compile time.
10.	It employs true and false values.	It employs YES and NO values, as well as BOOl.
11.	Swift has more template types than Objective-C.	Objective-C has lesser templates than Swift

ADVANTAGES OF OBJECTIVE-C

- The language was built as a preprocessor GCC package for current C compilers.
- C++ is more compatible with Objective-C.
- Objective-C has more stability.
- Private API use is simplified in Objective-C.

NEGATIVES OF OBJECTIVE-C

- Contrary to C++, Objective-C does not enable operator overloading.
- The original version of Objective-C did not enable garbage collection but still utilizes a C-based runtime, which increases the application's size.

- Since it is based on C Compilers, it requires a header file to function properly.
- Objective-C is a rather complicated language, although this is anticipated given its antiquity.

SHOULD WE GO FOR OBJECTIVE-C?

As can be seen, even a cursory comparison of the primary features of both languages reveals that Swift is superior to Objective-C. Moreover, one of Objective-C's last advantages over Swift has lately been eliminated.

Swift 5, the most recent version of the programming language, has ABI stability, a significant step toward attaining outstanding performance and security. Swift's lack of ABI stability was a significant issue until recently, but it has been addressed.

With the release of the newest version of Swift, Apple is pushing developers to utilize it for future iOS applications. The youthful language has a lot of unique benefits over the older and more demanding Objective-C, so it makes sense.

WHY SHOULD WE LEARN OBJECTIVE-C IN 2022?

Most macOS, XCode, and iOS kernel codes are written in C and C++. Since Objective-C is officially not a programming language but rather a huge preprocessor for pure C, it "fits in" with them rather well.

Objective-C is the foundation of Apple's whole ecosystem. Swift is mainly built on top of it. Therefore it is necessary to know the "base" to understand how the system works from the inside and why Swift looks and functions as it does.

Objective-C helps in understanding, for instance, that not all NSProxy objects are NSObject and the distinction between Int, NSInteger, and NSNumber. In addition to how Swizzling works, what a Selector is, how the responder chain operates, etc. This is strongly enclosed and abstracted in Swift, so it is not readily apparent.

In addition, like any C language, Objective-C facilitates an understanding of the fundamental operation of links, pointers, and memory.

Objective-C Is Still Used in the Development

First, it seems that every project that has lasted for more than 2–3 years has Objective-C code. It may be at least a hidden layer that is made reliant

and does not truly fit into the common code base, but it exists and requires periodic maintenance: bug fixes and feature additions.

While certain Apps cannot be rebuilt in Swift, doing so would involve time and money that neither the firm nor the client is willing to spend. Consequently, the Objective-C database is expanding.

Thirdly, if complicated work with network, memory, and device resources is required, languages with a lower level of abstraction than Swift are used. Depending on the amount of abstraction required, it may be Objective-C, Objective-C++, or just C and C++.

Swift is More Complex Than Objective-C for Some Tasks

Swift problem-solving may be tedious and tricky; using long-established Objective-C crutches is more convenient.

Primarily, we discuss memory management: to insert a message into unauthorized memory, one must actively control memory management or threads. Let us examine some instances.

Memory allocation, handling pointers and links directly, etc., are examples of Objective-C's purely "sish" aspects. Using Objective-C, you may thus gently optimize the program and increase speed and memory estimations when you must work very carefully with memory (such as when dealing with video and audio streams). You must use ARC while programming in Swift (although there are also life hacks here).

Furthermore, ARC does not resolve several issues, such as the lifetime of an object. Consider that there is a 14-minute voicemail. The customer has already listened to 12 – and they "ate up" the entire RAM. In Objective-C, they may be readily purged at the level of pointers and bytes, leaving just the current minute and the two preceding seconds. And on Swift, this will need a high degree of abstraction and a significant amount of code (or using Objective-C tricks via the Swift interface).

C++ is also used to create libraries that implement video or image recognition, computer vision, and cryptographic computations. And connecting with them through Swift is complex, costly, and time-consuming. Typically, it is simpler to build a header file in Objective-C, write a few methods for a nice wrapper, and refer to them immediately. Such occurrences are prevalent when connecting with third-party libraries.

OBJECTIVE-C LANGUAGE CHARACTERISTICS

We should be experienced with reading Objective-C code and mentally understanding how the runtime functions. This section examines the

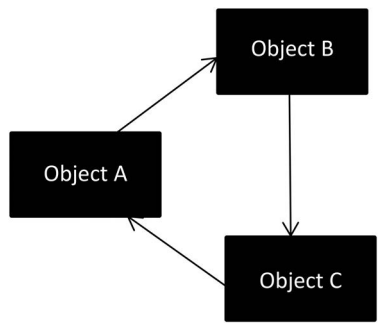
language and its semantics in further detail; it deconstructs the mental model we've constructed based on our intuition and exposes the principles that comprise Objective-C.

Strong and Weak Citations

A significant portion of programming Objective-C involves handling the references that running code has to objects and the references that objects hold to one another. Before introducing ARC, Objective-C used manual memory management on iOS and garbage collection on OS X. Both of these strategies have their benefits and drawbacks. Now it employs ARC, which incorporates the most advantageous elements of both approaches.

Automatic Reference Counting, or ARC, is a compile-time system that inserts manual memory management calls into the code as it is constructed. The compiler is quite intelligent, and the ARC code is well optimized. While ARC has liberated developers from writing manual memory management code, it cannot break reference cycles like garbage collection can-based platforms. For this reason, developers must instruct the compiler on how to prevent reference cycles.

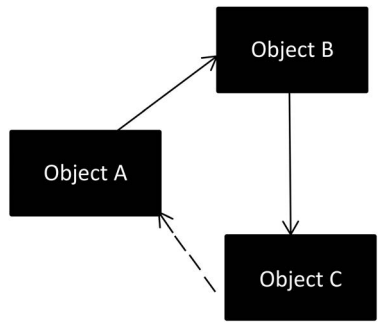
When an item no longer receives strong references, it is deallocated. If it was the last object with a solid reference for another object, that item is likewise deallocated. The diagram depicts a reference cycle.



Reference cycle.

Object A cannot be deallocated until Object C is, and Object C is not deallocated so long as Object B maintains a strong reference. Object B will never be released until Object A has been released. This is a reference cycle, which will lead to memory leaks in an application.

In below figure, the strong connection between Object C and Object A (represented by a solid line) has been replaced with a weak connection (symbolized by a dashed line).



Breaking reference cycle.

Weak references do not prevent objects from being deallocated. Therefore, Object A, Object B, and Object C are deallocated in that order.

If we have experience with C or C++, we will likely recognize an issue with weak references. If an object may be deallocated and we have a weak reference to it, a dangling pointer may result. With ARC compiling for iOS 5 and OS X 10.7, weak references to objects become null upon deallocation; these are known as zeroing weak references.

iOS 4 and OS X 10.6 do not support the weak keyword used to identify weak references. Hence unsafe_unretained must be used in its place. As its name suggests, it is risky to depend on a reference to an unsafe_unretained object. As long as we know this restriction, the application code we create should be safe.

Let's quickly explore the various sorts of Objective-C variables. Local variables belong to their enclosing scope (simply: the surrounding pair of curly brackets) and are only accessible during the execution of that scope. Additionally, there are instance variables, which are valid until the object instance is deallocated.

Local and instance variables are both strong references by default. Properties declared using the @property directive are likewise by default strong. The weak keyword (or, when declaring properties, the weak keyword in the qualifier list) is used to mark variables as weak.

The compiler determines how objects are returned from methods based on their names. Alloc, copy, init, mutableCopy, and new methods return objects that pass ownership to the calling procedure. All other methods

return objects without transferring ownership. The compiler uses this naming convention to insert the relevant memory management code.

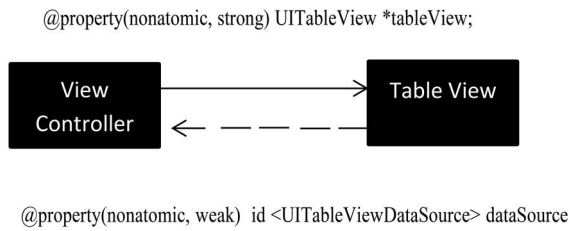
```
id instanceVariable;
...
-(void)performLongRunningTask
{
//ARC transfers ownership from copy to local variable
id obj = [someObject copy];

//ARC makes new strong reference to obj id anotherObj
= obj;

//Transferring to instance varable does not require
any memory management code
//instanceVariable is strong and init returns strong
reference instanceVariable = [[SomeObject alloc]
init];

} //anotherObj goes out of the scope, and its
reference is removed
```

ARC uses many optimizations to minimize unnecessary memory management code, so the code here is not an exact reflection of what occurs behind the scenes. Nonetheless, this is the mental picture you should establish as a developer.

Commonly, strong and weak references are "owned" and "owned by," respectively. When an item has another object, it employs a strong reference. If an item is owned by another object and has a reference to its owner, the reference is weak. This is not adequate reasoning for all references, but it is a helpful method for remembering the memory management principles of Objective-C.

As a specific example, let's examine NSArray. Arrays hold robust references to every element they include. We may consider an array to be the owner of these things, even if other objects may also own them.

The Delegation pattern in Objective-C provides a further illustration. Delegation is a typical technique for using loosely connected callbacks, particularly in user interface programs. A typical example of delegation in iOS is the UITableViewDataSource protocol. View controller-owned table views must know what to present to the user. The view controller must send this information to the table view.

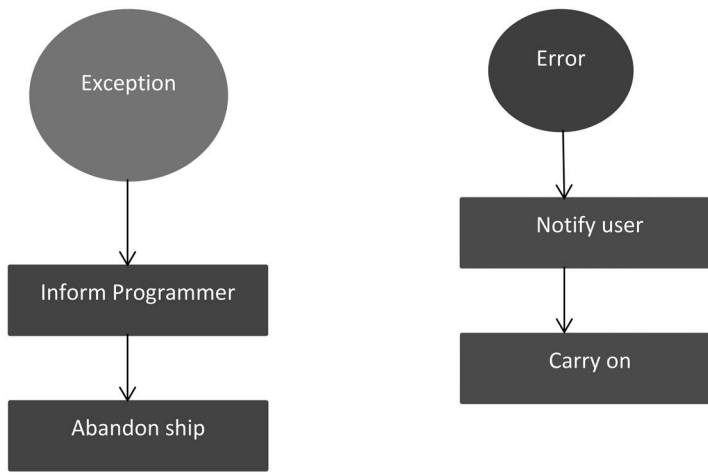


Weak references in the UITableViewDataSource.

OBJECTIVE-C: EXCEPTIONS AND ERRORS

There are two sorts of mistakes that can occur when a program runs in Objective-C. Unexpected errors are "serious" programming errors that cause your program to terminate prematurely. These are known as exceptions because they indicate a rare occurrence in your software. On the other hand, expected errors in Objective-C occur naturally throughout the execution of a program and can be utilized to evaluate the success of an activity. These are known as errors.

We might also think of 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.

For example, attempting to access a non-existent array index is an exception (a programming error), but failing to open a file is an error

(a user error). In the former situation, something went severely wrong in our program's flow, and it should presumably shut down immediately after the exception. In the latter case, we'd want to inform the user that the file couldn't be opened and potentially request that they attempt the operation, but there's no reason our software couldn't continue to operate after the error.

EXCEPTION HANDLING

The primary advantage of Objective-C exception handling features is isolating error detection from error treatment. When a piece of code detects an exception, it may "throw" exception to the closest error-handling block, which can "catch" and correctly handle particular errors. The ability to throw exceptions from random places avoids the need to continuously check for success or failure notifications from each function engaged in a given job.

@try, @catch(), and @finally are used to catch and handle exceptions, whereas @throw is used to detect them. If we have dealt with exceptions in C#, you should be acquainted with these exception-handling constructs.

It is crucial to realize that exceptions in Objective-C are pretty sluggish. Therefore, their usage is restricted to detecting severe programming flaws and not for fundamental control flow. If you are attempting to identify what to do in the event of an anticipated problem (e.g., failure to load a file), please refer to the section under Error Handling.

THE CLASS NSException

Instances of the NSException class or a subclass thereof represent exceptions. This is a simple method for encapsulating all the pertinent information about an exception. The three characteristics of an exception are as follows:

- name – An instance of NSString identifying the exception uniquely.
- reason – An instance of NSString that describes the exception in a human-readable format.
- userInfo – An instance of NSDictionary containing application-specific data about the exception.

The Foundation framework defines several variables defining "standard" exception names. These strings are used to determine the kind of caught exception.

We can also construct new exception objects with custom data using the initWithName:reason:userInfo: initialization method. Custom exception objects may be caught and thrown using the same techniques detailed in the following sections.

MAKING EXCEPTIONS

Let's begin by looking at a program's default exception-handling behavior. When we try to access an index Objective-C that does not exist, the objectAtIndex: function of NSArray throws an NSRangeException (a subclass of NSException). So, if we request the tenth item in an array with just three components, we'll have an exception to play with:

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

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

        NSArray *crew = [NSArray arrayWithObjects:
                         @"Shave",
                         @"Lywood",
                         @"Frank", nil];

        // throw an exception.
        NSLog(@"%@", [crew objectAtIndex:10]);

    }
    return 0;
}
```

When an uncaught exception is encountered, Xcode stops the application and shows you the line that caused the problem.

```
NSLog(@"%@", [crew objectAtIndex:10]); thread1: signal
SIGABRT
```

CATCHING EXCEPTIONS

Any code that may result in an exception should be placed in a @try block to manage it. The @catch() directive is then used to capture particular exceptions. If we need to run any housekeeping code, we may do so in a

@finally block. All three of these exception-handling directives are illustrated in the following example:

```
@try {
    NSLog(@"%@", [crew objectAtIndex:10]);
}
@catch (NSException *exception) {
    NSLog(@"Caught an exception");
    // We will just silently ignore exception.
}
@finally {
    NSLog(@"Cleaning-up");
}
```

The [crew objectAtIndex:10] message causes the software to throw an NSRangeException, captured by the @catch() command. The real exception handling occurs inside the @catch() clause. In this instance, we just show a descriptive error notice, but we'll likely need to create code to address the issue in other situations.

When an exception is detected in the @try block, the program goes to the matching @catch() block, preventing the execution of any code after the exception. This is problematic if the @try block requires cleanup (e.g., if it opened a file, it needs to be closed). The @finally block resolves this issue since its execution is assured regardless of whether an exception was thrown. This makes it the ideal location to resolve any outstanding issues from the @try block.

After the @catch() directive, the parenthesis allows you to specify the kind of an exception to capture. The exception is an NSException, the basic exception class in this instance. However, an exception may be any type, not simply NSException. The following @catch() directive, for example, will handle a generic object:

```
@catch (id genericException)
```

EXCEPTION THROWING

When an exceptional circumstance is detected in our code, we create an instance of NSException and populate it with the necessary details. Then, using the appropriately titled @throw directive, we throw it, causing the nearest @try/@catch block to handle it.

The following example creates a function for generating random integers between a defined interval. The function throws a custom error if the caller supplies an improper 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 minimum parameter"
            userInfo:nil];

        // Throw exception.
        @throw exception;
    }
    // Return a 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(@"The Random Number is: %i", result);

    }
    return 0;
}
```

This code will not throw an error since it supplies a valid interval (0, 10) to generateRandomInteger(). Change the interval to something like (0, –10), and we'll see the @catch() block in action. When the framework classes receive exceptions, this is what happens beneath the hood (e.g., the NSRangeException raised by NSArray).

We may also re-throw exceptions that you've already caught. This is handy if we want to be notified when an exception occurs but do not want to handle it ourselves. We may easily omit the argument to the @throw directive as a convenience:

```
@try {
    result = generateRandomInteger(0, -10);
}
@catch (NSException *exception) {
    NSLog(@"Problem! Caught exception: %@", [exception
name]);

    // Re-throw current exception.
    @throw
}
```

The caught exception is then sent up to the next-highest handler, which in this case is the top-level exception handler. This should show the output of our @catch() block and the standard Terminating program due to an uncaught exception … message, followed by an abrupt shutdown.

The @throw directive isn't restricted to NSException objects; it may throw anything. Instead of a standard exception, the following example throws an NSNumber object. Also, observe how we may target distinct objects by following the @try block with numerous @catch() statements:

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

int generateRandomInteger(int minimum, int maximum) {
    if (minimum >= maximum) {
        // Generate number using the "default"
interval.
        NSNumber *guess = [NSNumber
                           numberWithInt:generateRando
mInteger(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 default interval");
            result = [guess intValue];
        }
        @catch (NSException *exception) {
            NSLog(@"Problem! Caught exception: %@",
[exception name]);
        }

        NSLog(@"The Random Number: %i", result);

    }
    return 0;
}
```

The generateRandomInteger() attempts to create a new integer between specified "default" boundaries rather than producing an NSException object. The example demonstrates how to use @throw with various objects, but it isn't the ideal application architecture, nor is it the most efficient use of Objective-C exception-handling facilities. If we truly want to use the thrown value, as the preceding code does, we would be better served with a simple conditional check using NSError, as detailed in the next section.

Furthermore, several of Apple's core frameworks anticipate an NSException object to be thrown, so use caution when integrating with the standard libraries.

ERROR HANDLING

Errors are intended to be an efficient and easy method to determine if an activity was successful or not, while exceptions are intended to alert programmers when something has gone wrong. In contrast to exceptions, errors are intended to be utilized in routine control flow statements.

THE NSError CLASS

The only similarity between mistakes and exceptions is that they are both implemented as objects. The NSError class incorporates all information required to indicate errors:

- code – An NSInteger that provides a unique identification for the error.
- Domain – An instance of NSString that defines the error's domain (described in more detail in the next section).
- userInfo – An instance of NSDictionary containing application-specific data about the problem. This is used far more often than the userInfo dictionary of NSException.

In addition to these essential characteristics, NSError contains many variables that help in displaying and processing errors. These are all shortcuts into the userInfo dictionary given in the preceding list.

- localizedDescription – An NSString holding the error's whole explanation, often including the failure's cause. Typically, this figure is shown to the user via an alert panel.
- localizedFailureReason – An NSString offering an independent explanation of the error's cause. This is only used by customers that want to separate the error's cause from its entire explanation.
- recoverySuggestion – An NSString provides the user with instructions on how to recover from the mistake.
- localizedRecoveryOptions – An NSArray of titles for the error dialog's buttons. A single OK button to dismiss the warning is presented if this array is empty.

- helpAnchor – An NSString to show when the Help anchor button is pressed in an alert panel.

Similar to NSException, the initWithDomain:code:userInfo function may be used to initialize NSError objects with specific data.

ERROR DOMAINS

An error domain is essentially a namespace for error codes. Codes must be distinct within a single domain, although they may overlap with codes from different domains. In addition to avoiding code clashes, domains give information about the source of an issue. NSMachErrorDomain, NSPOSIXErrorDomain, NSOSStatusErrorDomain, and NSCocoaError Domain are the four primary built-in error domains. The NSCocoaError Domain includes error codes for most of Apple's standard Objective-C frameworks; however, some frameworks define their error domains (e.g., NSXMLParserErrorDomain).

If we need to generate new error codes for your libraries and Apps, we should always add them to your error domain; you should never expand any predefined error domains. Creating our domain is a simple endeavor. Because domains are just strings, we need to specify a string constant that does not clash with any of the application's other error domains. Apple recommends that domains use the form of com.<company>.<project>. ErrorDomain.

DETECTING ERRORS

There are no language constructs devoted to handling NSError objects (though several built-in classes are designed to handle them). They are intended for use with functions that return an object when successful and nil when unsuccessful. Following is the standard approach for capturing errors:

- Declare a variable named NSError. It does not need allocation or initialization.
- Pass this variable as a double pointer to a potentially error-prone function. If anything goes wrong, the function will utilize this reference to log error-related information.
- Check the function's return value for success or failure. If the action was unsuccessful, we may use NSError to handle or show the problem.

A function does not typically return an NSError object; instead, it returns the expected result if it succeeds and nil otherwise. Always use the function's return value to discover problems; never use the existence or absence of an NSError object to determine whether an operation was successful. Error objects are intended to describe a possible error, not indicate if one happened.

First, we create a file path to ~/Desktop/SomeContent.txt. Then, we construct an NSError reference and provide it to the stringWithContents OfFile:encoding:error: method to record any issues when loading the file. We've sent a reference to the *error pointer, which indicates that the function is asking a pointer to a pointer (i.e., a double pointer). This allows the method to supply the variable with its content. Finally, we examine the return value (rather than the existence of the error variable) to determine if stringWithContentsOfFile:encoding:error: succeeded or failed. If it does, we may work with the content variable's value; otherwise, we utilize the error variable to show information about what went wrong.

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

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

        // Generate desired file path.
        NSString *filename = @"SomeContent.txt";
        NSArray *paths =
NSSearchPathForDirectoriesInDomains(
                                NSDesktopDirectory,
NSUserDomainMask, YES
                            );
        NSString *desktopDir = [paths
objectAtIndex:0];
        NSString *path = [desktopDir
                             stringByAppendingPathCompone
nt:filename];

        // Try to load the file.
        NSError *error;
        NSString *content = [NSString
stringWithContentsOfFile:path

encoding:NSUTF8StringEncoding
                                error:&error];
```

```
        // Check if it is worked.
        if (content == nil) {
            // Some kind of the error occurred.
            NSLog(@"The Error loading file %@!",
path);
            NSLog(@"Description: %@", [error
localizedDescription]);
            NSLog(@"Reason: %@", [error
localizedFailureReason]);
        } else {
            // Content the loaded successfully.
            NSLog(@"Content-loaded!");
            NSLog(@"%@", content);
        }
    }
    return 0;
}
```

This code will likely result in an error as the ~/Desktop/SomeContent.txt file does not exist on our computer. Create a SomeContent.txt file on our desktop for the load to succeed.

CUSTOM ERRORS

We may set custom errors by taking a double reference to an NSError object and filling it with your data. Remember that your function or method should return an object or nil depending on its success (do not return the NSError reference).

The following example utilizes an error rather than an exception to handle erroneous arguments sent to the generateRandomInteger() method. Observe that **error is a double-pointer, allowing us to initialize the underlying value inside the method. It is crucial to verify that the user has given a genuine **error argument using if (error!= NULL). This is done in functions that generate errors. Since the **error parameter is a double-pointer, the *error parameter may assign a value to the underlying variable. Again, we check for mistakes using the return value (if (result == nil)) rather than 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:NSLocalizedDescription
Key];

            // Populate the error reference.
            *error = [[NSError alloc]
initWithDomain: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 what is went wrong.
            NSLog(@"An 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;
}
```

NSError's localizedDescription, localizedFailureReason, and associated attributes are kept in its userInfo dictionary using special keys provided by NSLocalizedDescriptionKey, NSLocalizedFailureReasonErrorKey, and so on. So, to characterize the issue, we just add some strings to the proper keys, as demonstrated in the previous sample.

Generally, we should provide constants for specific error domains and codes to ensure consistency between classes.

THE OBJECTIVE-C LANGUAGE'S FEATURES

Here is a shortlist of factors that, in my opinion, make Objective-C a strong and enjoyable programming language.

CLASSES ARE OBJECTS

Each class in Objective-C is an instance of a meta-class produced and handled automatically by the runtime. It is possible to create class methods, pass classes as arguments, store them in collections, etc. We just send a message to the class in Objective-C we want to instantiate to generate an instance. There is no need to redesign a "factory" system. No particular constructor mechanism is required at the language level. This helps to keep the language simple and effective.

Moreover, meta-classes are also objects!

DYNAMIC TYPING

Similar to Ruby, Python, Smalltalk, and Groovy… Extremely beneficial since we do not always know in advance what our runtime objects will be. Using dynamic typing in Objective-C is straightforward. This, for instance, defines a variable that may store a reference to an object:

```
id myObject;
```

OPTIONAL STATIC TYPING

Nonetheless, Objective-C also supports static typing. The ideal of both worlds.

This defines a variable capable of storing a reference to an object of type NSView or a subclass:

```
NSView *myObject;
```

CATEGORIES

Categories allow us to design new methods and add them to classes whose source code we do not know (such as the standard Cocoa classes provided by Apple). This facilitates class extension without the need for subclassing. Extremely helpful for adapting existing classes to the specifications of frameworks we want to utilize or build.

MESSAGE SENDING

We communicate with things through messages. Typically, the recipient of a message has a method that directly corresponds to the message (i.e., that has the same name or, in Objective-C terms, the same selector). In this situation, the method will be called. Nevertheless, this is not the only potential conclusion. An object may also handle a message by forwarding it to another object, broadcasting it to many objects, introspecting it, applying custom logic, etc.

The Syntax for Expressing Messages

Objective-C's message patterns are similar to natural language phrases with gaps (prefixed with colons). When we create code to convey a message to an object, we fill in the blanks with real data to form a coherent statement. This expressing message is derived from Smalltalk and makes the code extremely expressive.

Sending a message to an ordered collection, for example, requesting it to insert a specified item at index 10:

```
[myCollection insert:myObject atIndex:10]
```

A message sending expression may be interpreted as a sentence, with the subject being the receiver and the message being the remainder of the sentence (e.g., an action that we want the recipient to perform): "myCollection insert myObject at index 10."

INTROSPECTION

Object introspection is simple. For example, we may ask an object for its class using the following syntax:

```
[myObject class]
```

Determine whether an object contains the method "foo":

```
[myObject respondsToSelector:@selector(foo)]
```

Request the signature of an object's method "foo":

```
[myObject methodSignatureForSelector:@selector(foo)]
```

Ask whether a class is a subclass of another:

```
[class1 isSubclassOfClass:class2]
```

DYNAMIC OPERATING TIME

Objective-C has a dynamic runtime. It permits the creation of messages at runtime, the dynamic creation of classes, the dynamic addition of methods to existing classes, the modification of method implementations, etc.

AUTOMATIC GARBAGE COLLECTION

The automated garbage collector works simultaneously with the application code on its thread. It employs a generational approach to increase efficiency by focusing on memory zones that are more likely to contain trash. It works for both objects and raw C memory blocks allocated using the NSAllocateCollactable() method and equivalent techniques. The malloc() operates as expected, enabling access to memory that the collector does not maintain.

The garbage collector is an opt-in service; our application may instead depend on a reference counting mechanism if we prefer not to utilize it. This system has an inventive approach for delayed release that significantly reduces the effort of human reference counting.

Note that the iPhone does not support the automated trash collector at the time of writing.

C INSIDE

Objective-C programming language is an object-oriented extension of the C programming language and a superset of C. This implies that the raw power of C is accessible and that C libraries may be used directly (there are quite a few!). In addition, this provides a symbiotic link between the language and the operating system since Mac OS X, a UNIX system, is written mainly in C and Objective-C for the higher-level portions.

C++ FLUENT

Objective-C is not simply a superset of C, but it can also interpret and execute C++ code. In this setup, the language is referred to as Objective-C++ and permits the blending of Objective-C and C++ code statements. It also permits the direct use of C++ libraries.

SIMPLICITY

The Smalltalk-inspired object system of Objective-C tends toward simplicity. Many characteristics that tend to make languages difficult (templates, overloading, multiple inheritances, etc.) are missing in Objective-C, which provides simpler programming paradigms by leveraging its dynamic nature.

ACCESS TO APPLE'S PRODUCTS

Each new version of Mac OS X via Objective-C, and now the iPhone OS, is loaded with exciting new capabilities that are accessed straight from Objective-C. This makes Objective-C substantially more enjoyable to use.

WORKING WITH OBJECTIVE-C FILES

In Objective-C, we looked at the NSFileHandle, NSFileManager, and NSData Foundation Framework classes in Working with Directories in Objective-C. We examined how the NSFileManager class, in particular, allows us to interact with directories in Objective-C. This chapter progresses from dealing with directories to going through the specifics of working with files utilizing all three of these classes.

OBTAINING AN NSFileManager REFERENCE

First, we must review the processes required to gain a reference to create an instance of the NSFileManager class.

The defaultManager is used to create an instance of the class. As an example: get a hold of the NSFileManager object instance:

```
NSFileManager *filemgr;
filemgr = [NSFileManager defaultManager];
```

Checking to See If a File Exists

The NSFileManager class has a fileExistsAtPath instance function that determines if a particular file already exists. The method accepts an NSString object providing the path to the file as an input and returns a boolean YES or NO result indicating the presence or absence of that file:

```
NSFileManager *filemgr;

filemgr = [NSFileManager defaultManager];

if ([filemgr fileExistsAtPath: @"/tmp/myfiles.txt" ]
== YES)
        NSLog (@"The File is exists");
else
        NSLog (@"The File does not found");
```

Comparison of Two Files' Contents

The contentsEqualAtPath function compares the contents of two files for equivalence. This method accepts the paths to the two files to be compared as parameters and returns a boolean YES or NO indicating if the file contents match:

```
NSFileManager *filemgr;

filemgr = [NSFileManager defaultManager];

if ([filemgr contentsEqualAtPath: @"/tmp/myfiles.txt"
andPath: @"/tmp/sales.txt"] == YES)
        NSLog (@"The File contents is match");
else
        NSLog (@"File contents don't match");
```

Checking in Objective-C If a File Is Readable, Writable, Executable, and Deleteable

The majority of operating systems include some kind of file access control. These often take the form of properties that govern the degree of

access each user or user group has to a file. As a result, it is not guaranteed that our software will have read or write access to a certain file and the necessary rights to delete or execute it. Using the isReadableFileAtPath, isWritableFileAtPath, isExecutableFileAtPath, and isDeletableFileAtPath methods is the easiest way to determine whether our application has a certain access permission. Each method accepts a single input in the form of the file's path and returns a boolean value of YES or NO. For instance, the following code fragment verifies if a file is writable:

```
NSFileManager *filemgr;

filemgr = [NSFileManager defaultManager];

if ([filemgr isWritableFileAtPath: @"/tmp/myfiles.
txt"]  == YES)
        NSLog (@"The File is writable");
else
        NSLog (@"The File is read only");
```

File Moving/Renaming

The moveItemAtURL function is used to rename a file (given proper permissions). This function gives a boolean YES or NO result and accepts as inputs the pathname of the file to be relocated, the destination path, and an optional NSError object with information about any issues found during the operation. This option is set to NULL if no error description information is required. This procedure will fail if the destination file path already exists.

```
NSFileManager *filemgr;

filemgr = [NSFileManager defaultManager];

NSURL *oldPath = [NSURL fileURLWithPath:@"/tmp/
myfiles.txt"];
NSURL *newPath= [NSURL fileURLWithPath:@"/tmp/
newfiles.txt"];

[filemgr moveItemAtURL: oldPath toURL: newPath error:
nil];
```

Making a File Copy

The copyItemAtPath function is used to copy files. Like the move method, this function takes as arguments the source and destination pathnames and an optional NSError object. The returned boolean result indicates whether or not the operation was successful:

```
NSFileManager *filemgr;

filemgr = [NSFileManager defaultManager];

if ([filemgr copyItemAtPath: @"/tmp/myfile.txt"
toPath: @"/Users/demo/newfiles.txt" error: NULL]
== YES)
        NSLog (@"Copy-successful");
else
        NSLog (@"Copy-failed");
```

Delete a File

The removeItemAtPath function deletes the file supplied by the path from the file system. The pathname of the file in Objective-C is to be deleted, and an optional NSError object is sent as parameters to the procedure. The operation's success is reported as usual in the form of a boolean YES or NO return value:

```
NSFileManager *filemgr;

filemgr = [NSFileManager defaultManager];

if ([filemgr removeItemAtPath: @"/tmp/myfiles.txt"
error: NULL]  == YES)
        NSLog (@"Remove-successful");
else
        NSLog (@"Remove-failed");
```

Making a Symbolic Connection

The createSymbolicLinkAtPath function makes a symbolic link to a specific file. This function accepts the symbolic link's path, the path of the file to which the link should point, and an optional NSError object. The following code, for example, generates a symbolic link from/tmpUsers/demo/myfiles21.txt to the pre-existing file /tmp/myfiles.txt:

```
NSFileManager *filemgr;

filemgr = [NSFileManager defaultManager];
```

```
if ([filemgr createSymbolicLinkAtPath: @"/tmp/
myfiles2.txt"
                withDestinationPath: @"/tmp/myfiles.
txt" error: NULL] == YES)
        NSLog (@"Link-successful");
else
        NSLog (@"Link-failed");NSFileManager *filemgr;

filemgr = [NSFileManager defaultManager];

if ([filemgr createSymbolicLinkAtPath: @"/Users/demo/
files1.txt"
                withDestinationPath: @"/tmp/myfiles.
txt" error: NULL] == YES)
        NSLog (@"Remove-successful");
else
        NSLog (@"Remove-failed");
```

Reading and Writing Files with NSFileManager

The NSFileManager class provides essential file reading and writing functionality. These features are reasonably restricted compared to the possibilities supplied by the NSFileHandle class, but they can still be helpful.

To begin, the contents of a file may be read and stored in an NSData object using the contentsAtPath method:

```
NSFileManager *filemgr;
NSData *databuffer;

filemgr = [NSFileManager defaultManager];

databuffer = [filemgr contentsAtPath: @"/tmp/myfiles.
txt" ];
```

After storing the contents of a file in an NSData object, the data may be written to a new file with the createFileAtPath method:

```
databuffer = [filemgr contentsAtPath: @"/tmp/myfiles.
txt" ];

[filemgr createFileAtPath: @"/tmp/newfiles.txt"
contents: databuffer attributes: nil];
```

<google>IOSBOX</google> In the preceding example, we transferred the contents of an existing file to a new file. However, we cannot control how much data is read or written, and we cannot add data to the end of an existing file. If the file/tmp/newfiles.txt had previously existed in the preceding example, it and any data it contained would have been replaced by the source file's contents. A more adaptable system is necessary. The Foundation Framework provides this in the form of the NSFileHandle class.

Managing Files Utilizing the NSFileHandle Class

The NSFileHandle class has a variety of methods intended to give a more sophisticated mechanism for interacting with files. This class is used to manage devices and network sockets, and files. In the following sections, we will examine some of the most frequent applications of this class.

Creating an Object of Type NSFileHandle

When a file is opened for reading, writing, or modification, an NSFileHandle object is produced (reading and writing). The fileHandleForReadingAtPath, fileHandleForWritingAtPath, and fileHandleForUpdatingAtPath functions do this. After opening a file, it must be closed using the closeFile method after we are through working with it. If an attempt to open a file fails, such as when a non-existent file is attempted to be opened for reading, these methods return nil.

The following code sample, for instance, opens a file for reading and writing and then closes it without actually modifying the file:

```
NSFileHandle *file;

file = [NSFileHandle fileHandleForWritingAtPath: @"/
tmp/myfiles.txt"];

if (file == nil)
        NSLog(@"Failed to open the file");

[file closeFile];
```

NSFileHandle File Offsets and Seeking

NSFileHandle objects keep track of the current location inside a file. This is known as file offset. When a file is opened for the first time, its offset is 0 (the beginning of the file). This indicates that all read or write operations

using the NSFileHandle methods will begin at offset 0 in the file. To conduct actions at various positions inside a file, such as appending data to the end, it is essential to the first search for the desired offset. Use the seekToEndOfFile function to relocate the current offset to the end of the file. For instance, seekToFileOffset enables us to define the specific place inside the file where the offset should be positioned. The last way to identify the current offset is the offsetInFile method. The offset is saved as an unsigned long to accommodate large file sizes.

The following example opens a file for reading and then uses a series of methods to shift the offset to various locations, printing the current offset after each change:

```
NSFileHandle *file;

file = [NSFileHandle fileHandleForUpdatingAtPath:
@"/tmp/myfiles.txt"];

if (file == nil)
        NSLog(@"Failed to open the file");

NSLog (@"Offset = %llu", [file offsetInFile]);

[file seekToEndOfFile];

NSLog (@"Offset = %llu", [file offsetInFile]);

[file seekToFileOffset: 30];

NSLog (@"Offset = %llu", [file offsetInFile]);

[file closeFile];
```

File offsets are an important component of working with files with the NSFileHandle class, so take the time to ensure we grasp the notion. It is difficult to predict where data will be read or written in a file without knowing the current offset.

READING DATA FROM A FILE

After opening a file and assigning it a file handle, the contents of that file are read from the current offset point. The readDataOfLength function reads a specified number of bytes from a file, beginning at the current

offset. For example, the code below reads 5 bytes from offset 10 in a file. The data read is returned in the form of an NSData object:

```
NSFileHandle *file;
NSData *databuffer;

file = [NSFileHandle fileHandleForReadingAtPath:
@"/tmp/myfiles.txt"];

if (file == nil)
        NSLog(@"Failed to open the file");

[file seekToFileOffset: 10];

databuffer = [file readDataOfLength: 5];

[file closeFile];
```

DATA SAVING TO A FILE

The writeData method writes the data in an NSData object to the file beginning at the offset point. In Objective-C, it should be noted that this does not insert data but rather replaces any existing data in the file at the relevant spot.

To see this in action, we must first create a file. Create a file called quickfox.txt in the/tmp directory, enter the following content, and save it:

The quick brown fox overtook the sluggish hound.

Next, we'll develop a program that opens the file for updating, navigates to position 10, and then adds some data there:

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

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

        NSFileHandle *file;
        NSMutableData *data;

        const char *bytestring = "black dog";

        data = [NSMutableData dataWithBytes:bytestring
length:strlen(bytestring)];
```

```
        file = [NSFileHandle
fileHandleForUpdatingAtPath: @"/tmp/quickfox.txt"];

        if (file == nil)
                NSLog(@"Failed to open the file");

        [file seekToFileOffset: 10];

        [file writeData: data];

        [file closeFile];

    }
    return 0;
}
```

FILE TRUNCATION

The truncateFileAtOffset method truncates a file at the supplied offset. When invoking this function, give an offset of 0 to remove the entire contents of a file:

```
NSFileHandle *file;

        file = [NSFileHandle
fileHandleForUpdatingAtPath: @"/tmp/quickfox.txt"];

        if (file == nil)
                NSLog(@"Failed to open the file");

        [file truncateFileAtOffset: 0];

        [file closeFile];
```

HOW TO WORK AS AN iOS DEVELOPER

An iOS developer is a technology professional who creates mobile applications. They produce Apps that people use daily by applying their expertise in the iOS operating system and computer programming languages. If we appreciate utilizing technology to build new things, we might want to consider a career as an iOS developer. This piece in Objective-C will look at some of the most typical job tasks of an iOS developer and their average compensation and the measures that may take to become one.

WHAT EXACTLY IS AN iOS DEVELOPER?

An iOS developer is someone who builds software for mobile operating systems. They employ computer languages to create, test, and optimize programs that users may easily download and use. iOS developers can work as part of a team of other software engineers and IT experts at a software firm or as freelancers.

WHAT DOES AN iOS DEVELOPER DO?

The primary Objective of an iOS developer is to design mobile Apps that are pleasant to users. While the day-to-day responsibilities of an iOS developer may vary, they are often responsible for accomplishing specific tasks. Among the responsibilities you might anticipate doing in this position are the following:

- Testing, optimizing, and debugging programs to guarantee quality
- Recognizing possible concerns and overcoming challenges
- Coding and designing application architectures
- Publishing programs for download by customers
- Continuously doing maintenance to enhance application performance
- Developing and implementing modifications to an application
- Working with other members of the team to design and create new features.

iOS DEVELOPER INCOME AND EMPLOYMENT FORECAST

The average annual income for an iOS developer is $115,556. This might vary depending on our degree of education, job experience, and skill set. The sector we work in, the size of the company we work for, and the cost of living in our area may all influence our salary. In addition to their yearly income, many iOS developers get employer-provided perks. The most prevalent advantages include:

- Health insurance
- Health savings account
- Remote work opportunities

- Stock options
- Paid time off
- Commuter assistance

The Bureau of Labor Statistics (BLS) projects that the employment of software developers, quality assurance analysts, and testers will expand by 22 percent from 2019 to 2029, which is much higher than the average rate of growth for all professions. According to the BLS, this rise may be attributable to the growing need for computer software.

HOW TO BECOME AN iPhone PROGRAMMER

Here are some measures we may take to begin our iOS development career:

1. **Pursue an undergraduate degree:** Although it is possible to become an iOS developer without a bachelor's degree, finishing a bachelor's program in software engineering, computer science, or mathematics will help us acquire the knowledge and abilities necessary to flourish in this field. Additionally, many businesses choose people with formal training in a related profession. We may study the following courses throughout our undergraduate education to prepare for a job as an iOS developer:
    - Mobile development
    - Microcontroller architecture
    - Data structures and algorithms
    - Web development
    - Networking
    - Introduction to operating systems
    - Embedded systems
    - Computer programming languages
    - Assembly language
2. **Take Swift and Objective-C classes:** As an iOS developer, you must have a solid grasp of Swift, SwiftUI, and Objective-C. Swift is a fundamental programming language that enables the creation of variables

and the writing of functions to develop professional Apps in SwiftUI, a popular iOS framework. These packages give us all the resources necessary to develop user-friendly Apps. Objective-C classes are also available to acquaint students with another popular iOS programming language.

Swift and Objective-C use the same iOS foundations, yet the languages are somewhat distinct. Many iOS developers opt to study Swift first because it is more straightforward and more popular, but exploring Objective-C might give you a more well-rounded candidate for employment. Start by searching for online Swift and Objective-C courses.

3. **Develop our software:** Creating our apps is one of the most effective methods to progress as an iOS developer. This will allow us to experiment with various ways and acquire practical experience working with various programming languages. Providing samples of the apps we've developed may help impress hiring managers and illustrate our degree of expertise.

   Install Xcode on our computer to apply your Swift, Objective-C, and other programming language expertise. Xcode in Objective-C is the integrated development environment (IDE) used by iOS application developers. This tool allows you to write code, design user interfaces, and publish our Apps for people.

CONNECTIONS, DATA, AND THE CLOUD IN NETWORKING

In today's age of ubiquitous mobile devices and constant Internet connections, networking is a fundamental issue. On iOS alone, many Apps use the network in one way or another, either by utilizing built-in mechanisms such as iCloud synchronization or GameKit's communication architecture or by communicating directly with Internet-based servers. In addition, customer expectations often conflict with the capabilities and constraints of wide-area mobile networks like EDGE, 3G, and HDPSA+, leaving you, the application developer, to optimize a high-latency, low-throughput network channel.

Fortunately, I have vast knowledge and expertise to contribute to this subject. This section presented the URL loading mechanism and asynchronous, synchronous, and synthetic-synchronous network access. We will learn the best practices for handling various forms of data transmitted via the network and how to handle system-independent binary data

effectively. We will also learn how to identify services on the network and establish your services and make their information accessible in the same way.

Probably most significantly, we will discover the reasoning behind the following network-centric programming fundamentals on the Mac and iOS:

- The main thread should never visit the network.
- When feasible, use asynchronous network primitives.
- Use streams for all data processing.
- Ensure that data processing and transport handling are ignorant of one other.

BASIC PRINCIPLES

While the data access APIs on the Mac and iOS are typically storage-agnostic, there might be significant differences between loading a file from a disc and reading it through the network. To reinforce the teachings of this section, we'll take a few moments to clarify these distinctions and provide a solid foundation for whatever we may learn about networking in the future.

With both network and local file access, several issues might affect the performance of our program. These fall into two major categories:

Latency in Objective-C refers to the length of time required to reach the desired resource. This often refers to the seek-time of the underlying storage media when discussing local drives. This refers to the round-trip time of a network command packet.

Throughput is the rate at which data may be sent to or retrieved from a resource. Typically, local drives have a fast throughput and may transfer hundreds of megabytes per second. This might range from tens of gigabytes per second to a few kilobytes per second on a network.

Bandwidth availability is also related to these two categories. For local storage, reads and writes often pass across a hardware bus, which offers a much greater throughput than hard drives. Therefore, unless many discs are used concurrently, the total throughput is likely to equal that of the physical media utilized by a specific process. This indicates that if we read or write to numerous devices simultaneously, we will likely not see a significant increase in the throughput of each device compared to if we handled each device individually.

However, on the network, available bandwidth is often far more limited, and a single activity may rapidly use all available bandwidth, saturating the connection. Therefore, conducting multiple processes in parallel would likely result in a slowdown of all activities as the available bandwidth is divided among them. In addition, there are other nodes, such as wireless access points and routers, between your device and the real resource. Any of them may have bandwidth issues and any number of other clients, and they may impose particular restrictions to prevent anyone client from using excessive bandwidth. Many popular access points and routers, for instance, only let four concurrent HTTP requests from a single client to a single destination resource; all others are denied until some of the first four are completed. Therefore, it is generally advantageous for HTTP client software to apply the same restriction internally to prevent several concurrent activities from spinning their wheels and using local resources like memory, CPU, and battery life.

In contrast, there is often minimal difference when requesting several resources from the same place through the network, although doing so with files on the same local disc might cause a significant slowdown. The seek time is the root source of delay in disk-based storage systems. When reading a single huge file, the disc will search the file once and then read a (mostly) contiguous stream of data. When multiple files are being read and written concurrently, the disk's read head must repeatedly travel back and forth across the magnetic platter to switch between the files, resulting in disc thrashing. These clunking and clicking noises are audible during moments of intense computer activity. Recent advancements in solid-state storage have led to the widespread use of solid-state drives (SSDs) over platter-based magnetic hard disc drives; therefore, the latency issue is far less of a concern in such storage systems. The iPhone employs solid-state storage.

NETWORK DELAY

The most common causes of latency in network transmissions are network congestion (which may occur at any point along the path) or inefficiency of the underlying transmission protocol. Both wired and wireless network connections employ variants of the Ethernet protocol. This protocol is responsible for identifying other devices on the local network and efficiently transporting data between them. However, mobile devices using the cellular network use a protocol such as GSM or HSDPA, which must sustain data throughput even when devices are traveling fast between

endpoints such as cell towers. Consequently, these protocols have substantial overhead compared to Ethernet. This is exacerbated because most Ethernet or Wi-Fi devices have throughputs in the tens or hundreds of megabits per second, but HSDPA+ and "4G"1 cellular networks are only starting to surpass the 15 Mbit threshold. Consequently, it is usual to delay up to a full second (or even two!) while establishing a connection via a cellular network, especially with slower protocols such as GSM and 3G.

If your application makes many connections, such as when retrieving data from a server using the HTTP protocol, the latency might be a significant issue. Imagine you must make around 20 distinct calls to get a complete list of 200 comments (the server returns 10 per page). By default, each call will establish a new connection, and each connection will have significant latency. If the latency is high, regardless of the actual throughput speed, this will take between 10 and 40 seconds, which is not a pleasant experience. If throughput is similarly sluggish, the situation worsens. I've seen an implementation of synchronizing data from such an API take over an hour when many calls are to be performed, with network delay accounting for over half of that time.

ASYNCHRONICITY

With so many possible problems awaiting that, we must interact with network resources in an entirely asynchronous manner. We cannot emphasize this point enough: it must always be asynchronous, regardless of the circumstances.

It is easy to convince oneself in Objective-C that something does not need asynchronous execution. Possibly we use the network seldom; perhaps we only transmit or receive minimal quantities of data. Maybe we merely transfer data and do not wait for a response, or we only do network activities in reaction to user interaction. We've heard all these justifications, but we have some bad news: the network will still catch us. For instance:

- We seldom use the network:If we do it seldom, the user is probably not anticipating it. Our user is more likely to be frustrated if our infrequent action causes the program to become unresponsive than if it were a regular occurrence since they cannot anticipate when it will occur.
- We only transmit tiny data packets: Little packets may travel relatively rapidly on networks with limited capacity, but network delay

disregards packet size. Establishing the connection may take so long that the program becomes unresponsive, even for a short time, which is a Bad Thing.

- We simply send messages; we never wait for a response: Even if we are not expecting a response, our send operation will not finish until the network stack gets confirmation that the data has been received successfully. We might employ a protocol with unguaranteed data transport, but for delivering user data, this is a terrible choice; we need to ensure the data arrives securely.
- It is always triggered by human input. Thus users have learned to anticipate a little wait: The user may alter their decision. If everything is performed synchronously, there is no way to abort an action asynchronously if the user determines it takes too long.

SOCKETS, PORTS, STREAMS, AND DATAGRAMS

The OS X and iOS networking systems use the BSD sockets technology. This makes a network connection seem similar to a file: data is delivered and received by writing to or reading from a normal file handle, and the kernel handles the rest. There are additional capabilities, but it is doubtful that we will need to examine them for our objectives. When dealing with network connections, the primary distinction is that neither data nor space to write is accessible on demand. Incoming data is not accessible until it has arrived over the transport medium, and outgoing data is temporarily kept in a system buffer until the system can transmit it over the wire. We examined NSStream and observed events and methods based on these concepts: data accessible to read and space available to write.

The file handle of a network connection is known as a socket. Once constructed, they may be used very identically to conventional file handles until an NSFileHandle object is wrapped around them. The primary distinction is the inability to search inside the data stream; it is purely an ordered-access system.

There are two fundamental types of sockets.

Stream sockets are meant to transmit vast volumes of ordered data between two ends of a connection. Typically, the underlying transport guarantees material delivery in the right sequence.

On the other hand, Datagram sockets do not give any delivery or ordering assurances. Each datagram is treated as an independent entity, and no effort is taken to guarantee that all datagrams follow the same path across

the network. Depending on each node's circumstances and travel, each datagram might take a different path.

Datagram sockets, typically utilizing the Unified Datagram Protocol (UDP) over the Internet Protocol (IP), are typically used for streaming media. A datagram from a specific location in the underlying data stream is expected to arrive at a specific time. Still, the data is reassembled without receiving every byte in order. If we've ever seen a live webcast, we've likely seen the speaker's picture replaced with fragments of an earlier frame, looking twisted and wrapped to follow the speaker's movement. This demonstrates the absence of datagrams. Occasionally, the video data offers a whole frame, and in between, the data consists simply of alterations to apply to the previous frame to form the current frame. When a whole frame is missing, the picture of the incorrect frame is warped to construct a new one, resulting in the aforementioned aesthetic effect.

Streaming sockets and Transmission Control Protocol (TCP) over IP are used for most network communication. TCP guarantees delivery at the expense of time-to-destination guarantees. If a problem occurs during the transfer, the receiving endpoint may verify that it got the packet correctly and request a resend if necessary.

The TCP protocol also guarantees that order is preserved even if some packets arrive properly and others do not; if packet A fails and packet B succeeds, the contents of packet B are held until the contents of packet A are successfully delivered.

TCP provides transmission windows as well. These let each endpoint transmit the amount of data it can effectively handle to the other. Consequently, unlike UDP (Unified Datagram Protocol), an endpoint never transmits data that the recipient cannot presently process. On the other hand, Datagram protocols are intended to send data as quickly as possible (or at least at a constant pace), enabling the receiver to simply ignore any packets that come before it can process them.

We shall use TCP/IP rather than datagrams in the following instances. Datagram protocols often need additional application-level work to effectively handle failures such as missing packets, which is outside the scope of this book.

The Internet Protocol sits underneath the TCP and UDP protocols. This is the origin of many items we may already recognize: IP addresses with dotted octets, such as 192.168.23.5, derive from IP protocol version 4. Each connection endpoint is made accessible by a port on the IP address, a 16-bit integer identifying a specific endpoint. There are a variety of standard

port assignments recorded with the appropriate authorities, so some services are often available through a well-known port number. HTTP, for instance, uses port 80, while HTTPS utilizes port 443. When provided over TCP/IP, Apple Filing Protocol utilizes port 548, and there are many more common assignments in use, representing a significant portion of the protocol's 65535 available ports.

The 32-bit addressing mechanism used by IPv4 only permits around 4.3 billion different addresses. Add to that the fact that significant areas of these addresses are restricted from general use exclusively as local-network private addresses or for other communication, such as multicast IP, and it's not unexpected that we've run out. Early in 2012, service providers picked up the last available IPv4 addresses. The solution is the current implementation of IPv6, the most recent standard version. This upgrade employs a 128-bit addressing method, which is predicted to outlast the lives of everyone living now by a significant margin. From the software's perspective, the only difference is how an address is given.

AUTHENTICATION

Using two techniques, the URL connection tells its delegate of an authentication challenge. First, it will determine if the built-in credential storage is used to automatically search up login information by invoking – connection. When the connection is initiated, ShouldUseCredentialStorage is true. The connection will presume that the response is YES if the delegate does not implement this method.

This value will be used to generate a default challenge response if an authentication challenge is received from a remote service. This answer may already include valid credentials if the credential storage is accessed. If implemented, the – connection:willSendRequestForAuthenticationChallenge: method of the delegate will provide data on the challenge and any relevant stored credential that the system will try to use to satisfy the challenge. This is the most common technique for adding credentials to the store for subsequent usage. Thus, we should typically implement it.

The second argument is an instance of NSURLAuthenticationChallenge, which contains all known information on the challenge. We may identify the authentication mechanism, the suggested credential (if one was accessible from the store or if a protocol supports a common "anonymous" credential), the number of prior unsuccessful tries, and more using this object.

For instance, we can access an NSURLProtectionSpace object that describes the domain against which the credential is applied; this typically refers to the host, port, and protocol used and (depending on the protocol) may also specify a realm to restrict the authentication's applied scope further. For example, a conventional HTTP connection would provide a protocol of HTTP, a host of www.apple.com, and a port of 80 (the default port for HTTP services) (the default port for HTTP services).

An example realm for HTTP may be a subfolder: presumably, most of the host is available, but accessing www.testsite.com/secure/ needs a credential. In this case, the realm would be "secure" since the credential would only provide access to objects inside the folder resource. The protection area also offers data such as the certificate trust chain for a secure connection and information on whether the credential will be transferred securely. We may use any of these to determine whether or not to return a valid credential to the server.

The essence of the issue is the generation of an NSURLCredential object. To authenticate against a service, we must generate a credential and present it to the sender of the authentication challenge, available through the -sender method. The authentication challenge sender is any object that conforms to the protocol NSAuthenticationChallengeSender, which defines the methods.

DIFFERENCE BETWEEN OBJECTIVE AND OTHER LANGUAGES

C AND OBJECTIVE-C

C Language	Objective-C
It is a language based on procedures. Problems are resolved sequentially.	The computer language Objective-C is an object-oriented language. It provides syntax and semantics to enable an object-oriented programming language. It does not, however, enable multiple inheritance properties.
C language may be referred to as Objective-C's subset	Objective-C is the superset of the C programming language. It also has classes and objects in addition to the C programming language.
The C language's pointers are susceptible to security threats.	Objective-C utilizes null pointers and is hence type-safe compared to C.
It is fundamentally a low-level language that is too similar to assembly language.	Objective-C is a high-level programming language that combines C with a small-talk messaging style.

(Continued)

C Language	Objective-C
The C programming language does not provide classes. Bjarne Stroustrup created the C++ programming language to bring object-oriented capabilities such as classes to the C programming language.	Objective-C is an object-oriented programming language that supports classes and a dynamic runtime.
It uses a top-down programming methodology.	It uses a bottom-up programming methodology.
Large program code is broken down into smaller chunks called functions in this programming language.	Large program code is broken down into smaller codes known as Objects and Classes in this programming language.
It only supports pointers.	Both pointers and references are supported.
Declaring variables at the beginning of a program is required.	Variables are specified anywhere in this programming language.
The C programming language doesn't provide exception handling.	Exception handling in Objective-C may be implemented using catch and try sections.
C prohibits the definition of functions with default parameters.	It permits function definitions with default parameters.
C cannot execute Objective-C code.	Objective-C is capable of executing C code.
C includes malloc(), calloc(), and free() for dynamic memory allocation and deallocation, respectively.	Objective-C includes the new and deleted operators for memory allocation and deallocation.
Data is not protected in the C programming language.	Using the notion of encapsulation, Objective-C security is implemented.
There is no facility for inline function.	It supports the function inline.
C does not permit the overloading of functions and operators.	Objective-C enables the overloading of functions and operators.
It is often referred to as function-driven language.	It is known as object-oriented programming.
Encapsulation, Data hiding, inheritance, polymorphism, and abstraction are not supported.	Objective-C's most essential characteristics are encapsulation, data hiding, inheritance, polymorphism, and abstraction.
The C language lacks support for templates.	Objective-C is compatible with templates.
It is beneficial for embedded services.	It is helpful for gaming, networking, etc.

C++ LANGUAGE AND OBJECTIVE-C

Sr. No.	C++	Objective-C
01.	C++ is a high-level, object-oriented, procedural, general-purpose programming language. It was referred to as "C with Classes." It is a C programming language extension.	Objective-C language is a general-purpose, object-oriented programming language that provides syntax and semantics for object-oriented programming and operates as a superset of the C programming language.
02.	Bjarne Stroustrup created it in 1980 at Bell Laboratories.	1980: Brad Cox and Tom Love of Productivity Products International design the product (PPI).
03.	It allows for multiple inheritances.	It is not compatible with multiple inheritances.
04.	Structs and classes are handled identically in C++.	Structs and classes are not handled identically in Objective-C.
05.	In C++, methods are used to invoke the implemented functionality.	In Objective-C, messaging is used to invoke implemented functionality.
06.	It is a typed static language.	It is a typed dynamic language.
07.	True, false, and bool are used.	It employs YES, NO, and BOOL.
08.	Standard Template Library is present.	It lacks template libraries.
09.	Evernote, LinkedIn, Opera, Microsoft, NASA, and Facebook are among the businesses that use C++.	Uber, Pinterest, Instagram, Slack, Instacart, and more businesses use Objective-C.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Objective-C Cheat Sheet

Assumptions

This cheat sheet assumes our project has Automatic Reference Counting (ARC) enabled. Enjoy

Class Header (.h)

```
#import  "HeaderFile.h"
@interface Class_Name : SuperClass {
         //declare the instance variables
         //  (optional)
}
// define-properties
// define-methods (including any
//         custom-initializers) @end
```

Class Implementation (.m)

```
#import "ClassName.h" @implementation ClassName
// synthesize properties (optional in
//     the Xcode 4.4+)
// implement-methods (including any
//     custom-initializers, and dealloc) @end
```

Defining Methods

```
-  (anytype)doIt;
-  (anytype)doItWithX:(anytype)x;
-  (anytype)doItWithX:(anytype)x andY:
(anytype)y;
```

DOI: 10.1201/9781003357506-8

Implementing Methods

```
-  (anytype)doItWithX:(anytype)x andY:(anytype)y {
        //Do something with x and y... return retVal;
}
```

Creating a Class Instance

```
ClassName * my_Class = [[ClassName alloc] init];
```

Calling a Method

```
[myClass doIt]; [myClass doItWithX:x];
[myClass doItWithX:x andY:y];
```

Defining Properties

```
@property (attribute1, attribute2) property_Name;
```

strong	Adds ref to keep the object alive
weak	Object can disappear, and become nil
assign	Normal assign, and no reference
copy	Make the copy on assign
nonatomic	Make not threadsafe, increase perf
readwrite	Create the getter&setter (default)
readonly	Create just getter

Synthesizing Properties

```
//Optional in Xcode 4.4+ @synthesize property_Name;
@synthesize property_Name=
        _myInstanceVariable;
```

Using Properties

```
[myClass setProperty_Name:x]; myClass.property_Name =
x; // alternative

x = [myClass property_Name];
x = test.property_Name; // alternative
```

Declaring Variables

```
anytype my_Variable;
```

Int	1, 2, 500, 10000
float double	1.5, 3.14, 578.234
BOOL	YES, NO, TRUE, FALSE
ClassName *	NSString *, NSArray *, etc.
id	Can hold ref to any object

Custom Initializer Example

```
-  (id)initWithParam:(anytype)param {if ((self =
[super init])) {
                self.property_Name = param;
      }
      return self;
}
```

NSString Quick Examples

```
NSString *personOne = @"Raj"; NSString *personTwo =
@"Sham"; NSString *combinedString =
       [NSString stringWithFormat: @"%@: Heyyy, %@!",
personOne, personTwo];
NSLog(@"%@", combinedString); NSString *tipString =
@"24.99";
float tipFloat = [tipString floatValue];
```

NSArray Quick Examples

```
NSMutableArray *array = [NSMutableArray
arrayWithObjects:
         personOne, personTwo, nil]; [array
addObject:@"Waldo"]; NSLog(@"%d items!", array.count);
for (NSString *person in array) {
         NSLog(@"Person: %@", person);
}
//Xcode 4.4+ alt: array[2] NSString *waldo =
        [array objectAtIndex:2];
```

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Index

C

D

G

H

I

J

K

L

M

N

O

W

X