A Primer of

Swift Programming Language

Swift 2.1 (Released December 2015)

Part 1: The Basics

Xiaoping Jia
Professor, School of Computing, DePaul University
January 2016

Contact the author:
Email: xjia@cdm.depaul.edu
Web: http://venus.cs.depaul.edu/xjia

Hello, Swift!

Introduction

The Swift programming language was introduced by Apple at the *World Wide Developer Conference (WWDC)* in June 2014, as an alternative programming language to Objective-C for developing applications for iOS and Mac OS X. For Apple developers, it has been a long-awaited and welcome upgrade to Objective-C.

Swift is a modern programming language designed from ground up with a rich set of well-designed features. It is the newest member of the large family of programming languages descending from C, which includes Java, C++, C#, and many others. Its syntactic style and basic rules are very similar to those of the languages in the same family. If you are familiar with Java, C++, or C#, you will find the Swift syntax to be quite familiar and natural, and many of the basic lexical and syntactic rules that you are familiar with will still be valid in Swift. The designers of Swift have made great efforts to make the syntax of Swift concise and readable.

[1]

Swift is designed as an alternative to Objective-C for developing applications for iOS and Mac OS X. For this reason, Swift and Objective-C share the same libraries and frameworks available on iOS and OS X. Swift is designed to be *interoperable* at runtime with Objective-C and C. All runtime libraries for Objective-C and C are accessible in Swift. However, Swift is a completely new language, whose syntax is *not backward compatible* with Objective-C. It does not carry any old baggages from its many predecessors. You will pleasantly enjoy the fresh "new car" smell of Swift.

[2]

The Swift language has been evolving since it is initial release. A number of significant improvements has been introduced since its initial release. Many changes are not compatible with the initial version released in June 2014. The latest release at the time of writing is Swift 2.1 Refer to the official references, *the Swift Programming Language*[1], and *Swift Overview*[2] for the full definition of the language, the latest changes, and the revision history.

This set of notes is updated to Swift 2.1 and Xcode 7.2 (both released in December 2015).

Hello, World!

> *The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages:*
>
> Print the words
>
> `hello, world`
>
> — K & R, *The C Programming Language* [3]

It is a time honored tradition, dating back to *K & R*, to get acquainted with a new programming language through a little program known as *Hello world*. Without further ado, the *Hello world* in Swift.

Program 1.1: *Hello, World!*

Print the words "Hello, world!" on the console.

```
print("Hello, world!")
```

This one-liner, consisting of a single *statement*, is a complete program in Swift. This statement calls a *function* named `print()`, and provides a *string literal* `"Hello, world!"` as the argument. There is no need to import any libraries, nor is it necessary to enclose the statement in a function or a class.

You may also have noticed the absence of a semi-colon (;) at the end of the statement. In Swift, the statement-terminating semi-colon becomes optional, when a statement ends with a line-break, i.e., at the end of a line.

In Swift, the smallest executable unit of a program is neither a class, nor a function. It is a statement. In addition to statements, a Swift program usually also includes declarations of *classes* and *functions*.

[3] Brian Kernighan and Dennis Ritchie, authors of the seminal book *The C Programming Language* (Figure 1). Dennis Ritchie (1941–2011) was the original designer and implementer of the C programming language. He, along with long-time colleague Ken Thompson at AT&T Bell Lab, invented the Unix operating system. Ritchie and Thompson received the *Turing Award* from the ACM in 1983, the *Hamming Medal* from the IEEE in 1990, and the *National Medal of Technology* from President Clinton in 1999.

Program 1.1 Output:

```
Output
Hello, world!
```

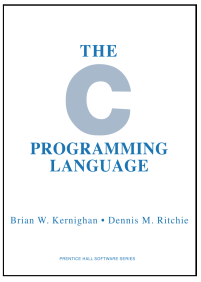


Figure 1: The cover of *The C Programming Language*, the First Edition, Prentice Hall, 1978. ISBN 0-13-110163-3. The Second Edition, Prentice Hall, 1988. ISBN 0-13-110362-8.

```
#include <stdio.h>

main()
{
  printf("hello, world\n");
}
```

Figure 2: The original *Hello world* in *K & R*

Swift API — Library: Swift (default)

Function:

`print(v)`

It prints the string representation of the value of the argument v to the standard output. A new-line is added to the end.

`print(v1, v2, v3, ...)`

The `print` function can take a variable number of arguments (at least one). It prints the string representations of the values of the arguments to the standard output, in the order they are given. Thee values are separated by a whitespace, and end with a new line.

Functions in Swift are declared in the global scope, and they are available everywhere in your program.

While you may have been impressed by the simplicity and succinctness of *Hello world* program in Swift, Swift goes a step further with full support of the latest version of Unicode [4]. Your greeting message can be just as easily in Chinese (or any other language), or in Emoji [5], as in English!

[4] *Unicode* is an international standard for a uniform digital representation of characters used in most writing systems in the world. The latest version of Unicode is Unicode 7.0, published in 2014. It contains over 113,000 characters.

[5] *Emoji*, n. (from Japanese) an ideogram or small image used to express an idea, emotion, etc., in electronic communications, especially in social media. Its use has been spreading rapidly in recent years. Most mobile platforms today support Emojis. In 2010, 722 Emoji characters were added to Unicode 6.0.

Program 1.2: *Hello, World!* in Chinese and Emoji

Print the words "Hello, world!" in Chinese and Emoji on the console.

```
print("你好, 世界!")
print("👋🌏❗")
```

Program 1.2 Output:

Output

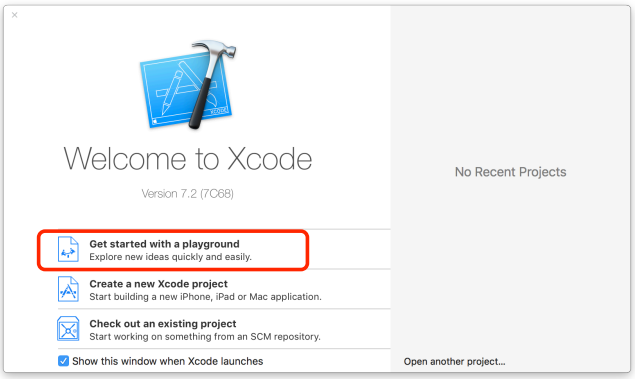
```
你好, 世界!
👋🌏❗
```

In Swift, characters from languages such as Chinese and Emojis are treated no differently from characters in Western languages. All are treated as characters, i.e., values of the `Character` type.

Swift development tools are fully supported only on MacOS X[6]. We will use the development environment Xcode and Playground available on MacOS X. A Mac computer running MacOS X 10.9.4 or later, i.e., OS X Mavericks (10.9.x), OS X Yosemite (10.10.x), or OS X El Capitan (10.11.x) is required. To compile and run Swift 2.1 programs, you must download and install Xcode 10.7 or later [7].

Xcode provides several tools for you to edit, debug, and run Swift programs[8]. The easiest way to get started is using a very nice tool introduced in Xcode 6.0 called *Playground*. Playground is an interactive execution environment for running Swift programs. It is ideally suited for learning the language and experimenting with various features. It even allows you to visualize the results and to experiment with graphical user interface (GUI) features designed for iOS and OS X.

To launch an Xcode Playground, you first launch the Xcode. At the Xcode launch dialog, select "Get started with a playground" (Figure 3) [9].



[6] In December 2015, apple made Swift development tools open source. In addition to MacOS X, Swift development tools now are also available on Linux, and possibly other platforms in the future. However, Xcode and Playground are still only supported on MacOS X. At the time of writing, only command-line based development tools are available on Linux. Frameworks for MacOS X and iOS are not available on Linux.

[7] We will use Xcode 7.2. The latest version of Xcode can be downloaded from `https://developer.apple.com/xcode/`

[8] In addition to using Playground, Swift programs can also run in Xcode projects or in a command-line tool – *Swift REPL*, stands for *Read Eval Print Loop*.

[9] If Xcode is already running, you can launch a new Playground from the Xcode menu bar:
File ▶ New ▶ Playground . . .
Shortcut: ⌥⇧⌘N

Figure 3: Launch an Xcode Playground. The launch screen of Xcode.

The next dialog will prompt you to choose (Figure 4):

- a name for your Playground file, and
- a platform: "iOS", "OS X", or "tvOS". You may choose any of the three for simple programs without UI.

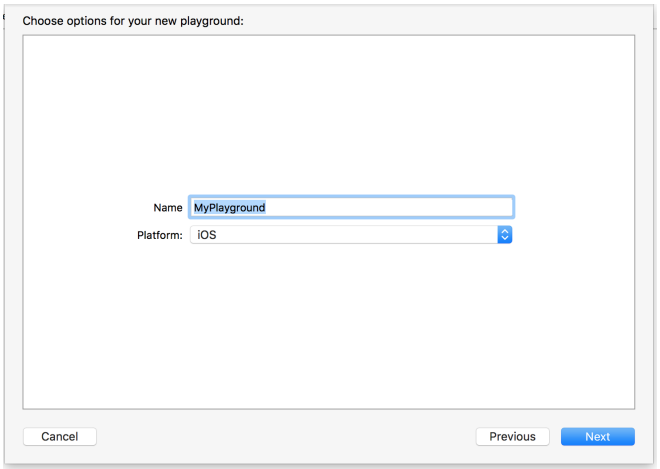


Figure 4: Launch an Xcode Playground. The dialog for choosing a name and a platform for a new Playground.

After filling in the choices, click "Next" to choose a folder to save the Playground file. The Playground files will be saved with the .playground suffix. Finally, click "Create" to create a new Playground.

The new Playground created is shown in Figure 5.

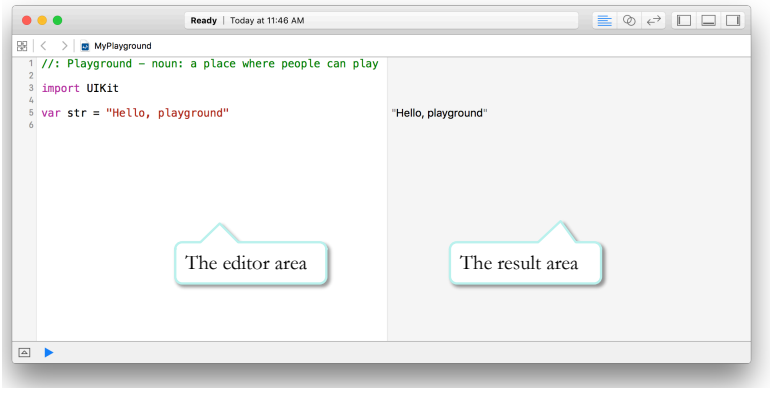


Figure 5: The Xcode Playground. A new Playground with an editor area and a result area.

The main area of the initial view consists of two side-by-side panels. The left panel is the *editor* area, which allows you to enter Swift code. The right panel is the *result* area, which displays the result of each corresponding line of code in the editor area. The results of your program are displayed and updated in the result area automatically and immediately after you have completed typing the code in the editor, and if the code contains no error.

PLAYGROUND STARTS WITH A SIMPLE TEMPLATE. The first line is a comment. The `import` statement imports the UIKit framework, the UI library for iOS [10]. The last line declares a variable, which we will discuss a little later. You can add your own code following these lines, and start experimenting. Enter the *Hello World* programs in the Playground editor, and you will see the results appear in the result area immediately (Figure 6).

[10] The `import` statement here is not needed for experimenting basic language features. For all the examples in this section, we only use the features in the Swift library, which is available by default. It is safe to remove the `import` statement here.

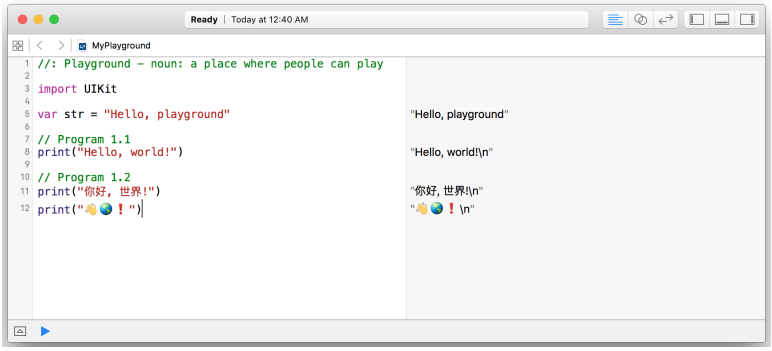


Figure 6: The Xcode Playground. The *Hello World!* program in the new Playground.

If you hover the mouse over the results in the result area, two additional controls will appear near the right edge of the result area. The control on the left is the *Quick Look* control, as shown in Figure. 7

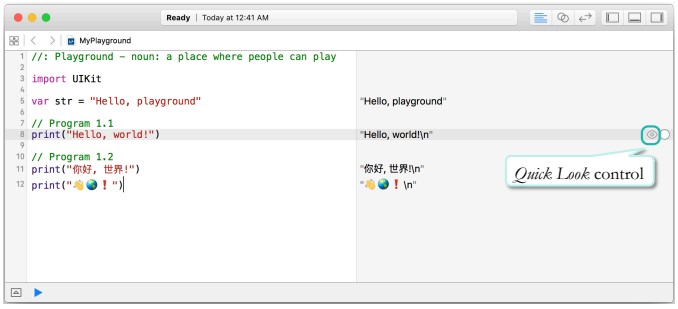


Figure 7: The *Quick Look* control in the result area of the Playground.

Click on the *Quick Look* control, the *Quick Look* popup will appear, which is convenient for results that requires a larger area to display.

Figure 8: The *Quick Look* popup.

When you hover the mouse over the results in the result area, the control on the right is the *Show Result* control, as shown in Figure. 9

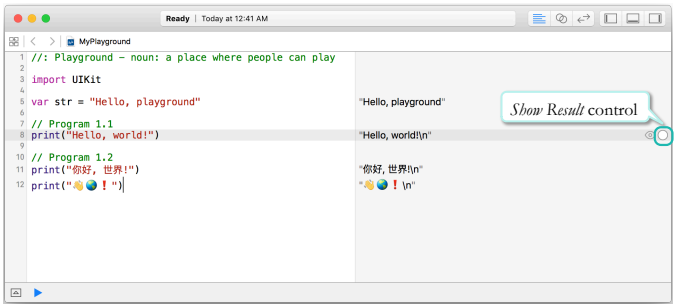


Figure 9: The *Show result* control in the result area of the Playground..

Click on the *Show Result* control, an *inline result* area will be insert into the editor area just below the corresponding line of the code. The results in the inline result area will be automatically updated after the code is modified and executed.

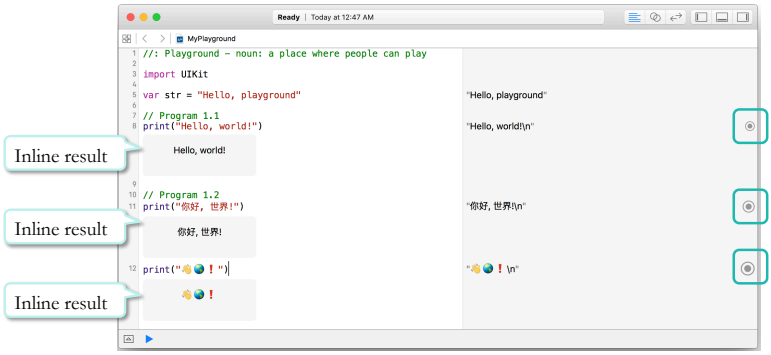


Figure 10: The inline results of the program in the Playground.

Another useful feature of the Playground is the *Debug Area* near the bottom, as shown in Figure 11.

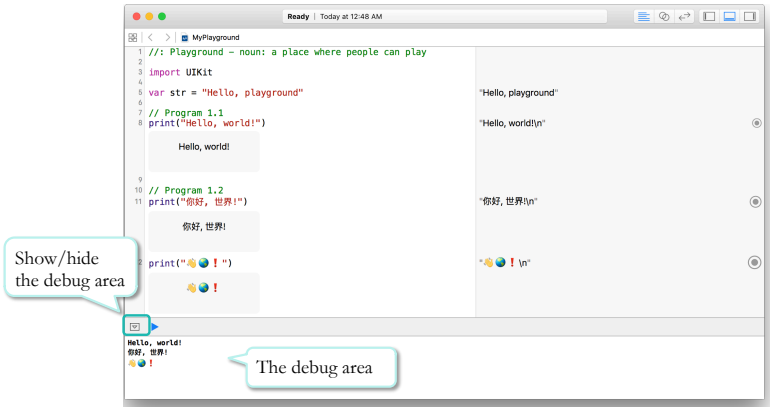


Figure 11: The *Debug Area* of the Playground.

You can use the control near the lower-right corner to show and hide the *Debug Area*. The debug area displays messages if your code

contains errors, and it displays the output to the standard output, i.e., the console, when your code executes.

A Playground may contain a full-fledged Swift program, which consists of any of the following language elements:

- *statements*, including *constant* and *variable declarations* and *expressions*,
- *function declarations*, and
- *class declarations*, etc.

We will discuss each of these elements in the following sections.

The Data Types

The Lexical Elements

Whitespace and Comments

Whitespace is insignificant in Swift code in most situations. Whitespace is ignored by the compiler, except to separate tokens.[11]

[11] However, in Swift, there is a slight complication. A whitespace adjacent to an operator could be significant, i.e., the presence or absence of the whitespace may alter the meaning of the adjacent operator.

Comments are non-executable elements in programs that are treated as whitespace by the compiler. They are useful for including documentations, notes, and reminders in the program. Swift provides two types of comments, which are similar to the comments in Java and C++.

- A *single-line comment* starts with `//` and ends at the end of the line.

```
// This is a single-line comment.
```

- A *multi-line comment* starts with `/*` and ends with `*/`. It may span across several lines.

```
/* This is a multi-line comment.
    It spans across several lines */
```

Unlike Java and C++, Swift allows multi-line comments to be nested, which makes it easy to comment out a large block of code that contains multi-line comments. For nested multi-line comments, the opening marks `/*` and the closing marks `*/` must be balanced.

* This icon will be used to indicate language features that are unique or different in Swift, which deserve some special attention.

```
/* The start of the outer multi-line comment.
    /* The start of the nested multi-line comment.  ...
       More nested comment.  ...
    */
   The end of the outer multi-line comment */
```

Identifiers

Identifiers are used as names of variables, constants, functions, and classes, etc., in your programs. The rules for forming identifiers in Swift are similar to the rules in Java or C++.

Definition: Identifier

- An *identifier* consists of a sequence of alphanumerical characters and the underscore (_) character.
- An identifier may not start with a digit.
- Characters not allowed in identifiers include:
  - punctuations, such as `;`
  - operators, such as `+ - *`
  - other special characters, such as `$`

The following are some examples of legal identifiers in Swift. [12]

`message`	`firstName`	`lastName`	`statusMessage`
`pi`	`sum`	`radius`	`totalPrice`
`x`	`y`	`t0`	`t1`
`x1`	`y1`	`x_1`	`y_1`
`count`	`count100`	`totalAttempts`	`maximumVelocity`
`move`	`next`	`increaseBy`	`calculateAverage`
`Student`	`Model10`	`SpaceShip`	`ComplexNumber`

[12] Swift also allows Unicode characters to be used in identifiers, excluding characters used for punctuations, operators, and other special characters, etc. The following are also legal identifiers in Swift.

π 🐶🐮 你好

It is always a good practice to choose descriptive names, whose meanings are self-evident within the context in which they are used. You may also have noticed that identifiers with multiple words are formed using *camel case* [13]. The following naming conventions are widely adopted in Swift. It is a good practice to consistently follow these conventions.

Naming Conventions

1. Names comprising of multiple words should be formed by joining the words using camel case.
2. Variable, constant, and function names should start with a lowercase letter.
3. Type names, including class names, should start with an uppercase letter.

[13] *Camel case*: A common method to form an identifier from multiple words by capitalizing each word, except the first, and then concatenating all the words without space between words. The first word may start with either an upper- or a lowercase letter.

Basic Data Types

Data types ARE THE BUILDING BLOCKS OF PROGRAMS. A data type is a set of values and a set of operations on those values. Swift provides a rich set of *basic types*, also known as *fundamental types*, for representing numbers, strings, characters, and Boolean values. The most commonly used basic types are the following:

Type	Values	Examples
`Int`	integer numbers	`2014`, `-32`
`Double`	floating-point numbers	`3.1415927`, `-100.0`
`String`	string values	`"Hello, World!"`
`Character`	single Unicode characters	`"A"`, `"t"`
`Bool`	Boolean values	`true`, `false`

Table 1: The most commonly used basic types in Swift.

SWIFT DOES NOT DIFFERENTIATE between two tiers of types – primitive types and object types. There is no distinction between system defined, or built-in, types and user-defined types either. Swift treats all types uniformly [14]. The basic data types in Swift are defined as *structs*, which have similar syntax and capabilities as classes, but with the overhead usually associated with classes and objects. The values of basic types are first class citizens, that the values of basic types can be used anywhere objects are expected with no conversion necessary, i.e., no boxing and unboxing necessary in Swift.

[14] Swift does differentiate between *value* types from *reference* types. We will discuss the differences in detail later. All basic types are value types.

Integer Types

SWIFT PROVIDES SEPARATE DATA TYPES for *signed integers*, which can be positive, zero, or negative, and *unsigned integers*, which can be positive or zero, in 8, 16, 32, and 64 bit forms. The singed integer types begin with the prefix `Int`, and the unsigned integer types begin with the prefix `UInt`. The integer types provided by Swift are summarized in Table 2 [15]

[15] The maximum and minimum values of each integer type are accessible in the program using the `max` and `min` property of the respective type. For example

```
Int.max     Int.min
UInt16.min  UInt16.max
```

Type Name	Description	Minimum Value	Maximum Value
`Int8`	8-bit signed	-128 (-2^7)	127 (2^7 - 1)
`Int16`	16-bit signed	-32,768 (-2^{15})	32,767 (2^{15} - 1)
`Int32`	32-bit signed	-2,147,483,648 (-2^{31})	2,147,483,647 (2^{31} - 1)
`Int64`	64-bit signed	-9,223,372,036,854,775,808 (-2^{63})	9,223,372,036,854,775,807 (2^{63} - 1)
`UInt8`	8-bit unsigned	0	255 (2^8 - 1)
`UInt16`	16-bit unsigned	0	65,535 (2^{16} - 1)
`UInt32`	32-bit unsigned	0	4,294,967,295 (2^{32} - 1)
`UInt64`	64-bit unsigned	0	18,446,744,073,709,551,615 (2^{64} - 1)

Table 2: The integer types and their ranges.

The type `Int` is a signed integer type, whose size matches the native word size of the target platform, i.e., [16]

- When the target platform is 32-bit, `Int` is the same as `Int32`.
- When the target platform is 64-bit, `Int` is the same as `Int64`.

The type `UInt` is an unsigned integer type, whose size matches the native word size of the target platform, i.e.,

- When the target platform is 32-bit, `UInt` is the same as `UInt32`.
- When the target platform is 64-bit, `UInt` is the same as `UInt64`.

[16] Both 32-bit and 64-bit processors are used in various models and editions of iOS devices. The original iPhone launched in 2007 used a 32-bit processor. So did the original iPad launched in 2011. The first 64-bit processor for iOS devices was the *Apple A7 Processor*, which was first used in iPhone 5S and iPad Air in 2013. The first version of iOS that supports 64-bit processors was iOS 7.0, released in September 2013.

A *literal* REPRESENTS a constant value of a type[17]. Integer literals are written in decimal notation by default. To make the large numbers easier to read, you may add underscores (`_`) to group digits in integer literals. The underscores will be ignored by the compiler and will not affect the underlying value of the literals. You may also pad `0` at the front of integer literals.

[17] In Swift, integer literals of different types are expressed in the same way. In Swift, literals are considered *typeless* or *convertible* to several different but related types. In other words, they do not have fixed types. Their types are determined based on the contexts in which they appear.

```
123             -123            12300
12_300          012_300         -012_300
1_000_000       1_234_567_890
```

Examples of integer literals in decimal notation.

SWIFT ALSO ALLOWS INTEGER LITERALS to be written in alternative notations using the following designated prefixes

- `0b` for binary notion (base 2)
- `0o` for octal notation (base 8)
- `0x` for hexadecimal notation (base 16)

```
        123          -123      Decimal notation
0b0111_1011  -0b0111_1011      Binary notation
      0o173        -0o173      Octal notation
       0x7B         -0x7B      Hexadecimal notation
```

Examples of integer literals in decimal, binary, octal, and hexadecimal notations. All of these integer literals have a decimal value of 123 or -123.

Arithmetic Expressions

Expressions in Swift are very similar to expressions in Java and C++. Expressions are formed using operators of various data types. *Arithmetic expressions* are binary expressions on numeric types using *infix* arithmetic operator [18]. In Swift, both operands of an arithmetic expression must be of the exact same type, and the result of the expression is of the same type as its operands. The operands can be literals or expressions. The usually rules of precedence and associativities of operators apply in Swift. Parentheses can be used in expressions where needed to override the order of precedence. The common arithmetic operators on integers are summarized in Table 3.

[18] An *infix* operator is an operator that takes two operands to form a *binary* expression, where the operator appears in between the two operands, e.g., `5 + 3`.

Operators	Expressions	Results
`+` (addition)	`5 + 3`	8
`-` (subtraction)	`5 - 3`	2
`*` (multiplication)	`5 * 3`	15
`/` (division)	`5 / 3`	1
`%` (remainder)	`5 % 3`	2

Table 3: Common arithmetic operators on integers.

Floating-Point Types

Swift provides the following signed floating-point number types:

Type Name	Description
`Float`	32-bit single-precision floating-point numbers
`Double`	64-bit double precision floating point numbers

Table 4: The floating-point types.

Floating-point literals are written in decimal notation by default. A float-point literal must have a *fraction* or an *exponent*, or both. A fraction consists of a decimal point (`.`) followed by a sequence of digits. If a fraction is present, there must be at least one digit on either side of the decimal point. An exponent begins with an upper- or lowercase `e` followed by a signed decimal integer. A decimal exponent exp multiplies the base number by 10^{exp}. To make the floating-pointing numbers easier to read, you may add underscores (`_`) to group digits in floating-point literals. The underscores will be ignored by the compiler and will not affect the underlying value of the literals. You may also pad `0`'s that are insignificant. The following are some examples of floating-point literals in decimal notation and their corresponding values.

`123.45`	123.45	`-123.45`	-123.45
`1.2345e2`	123.45 (1.2345×10^2)	`-1.2345E2`	-123.45 (-1.2345×10^2)
`1.2345E-3`	0.0012345 (1.2345×10^{-3})	`-1.2345e-3`	-0.0012345 (-1.2345×10^{-3})
`1E10`	10,000,000,000.0 (1×10^{10})	`-1E-10`	-0.0000000001 (-1×10^{-10})
`012_234.000_010`	12,234.00001	`-012_234.000_010`	-12,234.00001

Floating-point literals can also be written in hexadecimal notation with the prefix `0x`. An hexadecimal exponent begins with an upper- or lowercase `p` followed by a signed *decimal* integer. A hexadecimal exponent exp multiplies the base number by 2^{exp}. A hexadecimal floating-point literal must end with an exponent. The following are some examples of floating-point literals in hexadecimal notation and their corresponding decimal values.

`0x7p0`	7.0 (7.0×2^0)	`-0x7p0`	-7.0 (-7.0×2^0)
`0x7.Bp0`	7.6875 (7.6875×2^0)	`-0x7.Bp0`	-7.6875 (-7.6875×2^0)
`0x7.Bp2`	30.75 (7.6875×2^2)	`-0x7.Bp2`	-30.75 (-7.6875×2^2)
`0x7P8`	1,792.0 (7.6875×2^8)	`-0x7P8`	-1,792.0 (-7.6875×2^8)
`0x7p-2`	1.75 (7.6875×2^{-2})	`-0x7p-2`	-1.75 (-7.6875×2^{-2})
`0x0123_4567p8`	4,886,718,208.0	`-0x0123_4567p8`	-4,886,718,208.0

The common arithmetic operators on floating-point numbers are summarized in Table 5.

Operators	Expressions	Results
+ (addition)	`20.14 + 3.14`	23.28
- (subtraction)	`20.14 - 3.14`	17.0
* (multiplication)	`20.14 * 3.14`	63.2396
/ (division)	`20.14 / 3.14`	6.4140127388535
% (remainder)	`20.14 % 3.14`	1.3

Table 5: Common arithmetic operators on floating-point numbers.

String and Character Type

A *string* CONSISTS OF A SEQUENCE OF CHARACTERS, such as `"Hello, World!"` or `"你好，世界!"`. Strings are represented by the `String` type, which represents a sequence of Unicode characters. Characters are represented as values of the `Character` type.
A string literal consists of a sequence of characters surrounded by a pair of double quotes (`"`). An empty string literal is written as `""`. String literals cannot contain an unescaped double quote (`"`), an unescaped backslash (`\`), a carriage return, or a line feed. Special characters can be included in string literals using the escape sequences shown in Figure 12. You can also specify a character using its Unicode scalar value in the form of

`\u{`*n*`}`, where *n* is between one to eight hexadecimal digits.

Character	Escape Sequence
Null character	`\0`
Backslash	`\\`
Horizontal tab	`\t`
Line feed	`\n`
Carriage return	`\r`
Double quote	`\"`
Single quote	`\'`

Figure 12: Character escape sequences.

The following are some examples of string literals and their corresponding values.

```
"\"I'm sorry Dave, I'm afraid I can't do that.\" - HAL 9000"
```
"I'm sorry Dave, I'm afraid I can't do that." - HAL 9000

A quote from *2001: A Space Odyssey* (1968).

`"\u{A9}"`	© (the Copyright symbol)
`"\u{20AC}"`	€ (the Euro symbol)
`"\u{4F60}\u{597D}\u{4E16}\u{754C}"`	你好世界 (*Hello World* in Chinese)

THE OPERATOR + CAN BE APPLIED TO TWO STRING VALUES to perform a concatenation of the two strings [19]. For example, the following expression

```
"Hello, " + "world!"
```

yields the result: `"Hello, world!"`.

[19] In Swift, the string concatenation operator + requires both operands to be strings. In other words, no implicit conversion of non-string values to strings will be performed. For combining strings with values of different types use *string interpolation* (p. 26).

Boolean Type

Swift provides a `Bool` type to represent Boolean values, which are commonly used as conditions in control statements, such as `if` and `while` statements. Unlike C++ and Objective-C, the `Bool` type in Swift is distinct from integer types. In other words, integer values cannot be used where a Boolean value is expected. The `Bool` type has only two values: `true` and `false`.

A *logical expression* is an expression that its operands are Boolean values and its result is Boolean as well. The common infix logical operators && (and), || (or), as well as the prefix operator ! (not) are supported in Swift (Table 6). The logical operators && and || have the usual *short-circuit* semantics, as in C++ and Java[20].

[20] *Short-circuit* evaluation of Boolean expressions means:

a) When evaluating *a* && *b*, if *a* evaluates to `false`, *b* will not be evaluated, i.e., short circuited. The result of the expression is `false`.

b) Similarly, when evaluating *a* || *b*, if *a* evaluates to `true`, *b* will not be evaluated, i.e., short circuited. The result of the expression is `true`.

Operators	Expressions	Results
&& (and)	`true && false`	`false`
|| (or)	`true || false`	`true`
! (not)	`!true`	`false`

Table 6: Common logical operators on Boolean values.

A *comparison expression* is a binary expression that compares two values of the same type using one of the infix *comparison operators* and yields a Boolean value as the result. Swift supports all the common comparison operators: == (equal to), != (not equal to), < (less than), <= (less than or equal to), > (greater than), and >= (greater than or equal to). All comparison operators can be applied to all basic types.

```
5 == 3    20.14 == 3.14    "Swift" == "Objective-C"
5 != 3    20.14 != 3.14    "Swift" != "Objective-C"
5 < 3     20.14 < 3.14     "Swift" < "Objective-C"
5 <= 3    20.14 <= 3.14    "Swift" <= "Objective-C"
5 > 3     20.14 > 3.14     "Swift" > "Objective-C"
5 >= 3    20.14 >= 3.14    "Swift" >= "Objective-C"
```

Examples of comparison expressions on different types.

The equality and non-equality comparisons are performed based on the *values* of the operands [21]. The inequality comparisons on strings are based on the dictionary order, where the order of characters is based on the Unicode standard [22].

[21] Swift provides different comparison operators for comparing identities of values of reference types.

[22] The Unicode order is consistent with the alphabetical order of Western languages.

Boolean expressions are expressions whose results are Boolean values. Comparison expressions and logical expressions are both Boolean expressions.

Declarations

Programs manipulate *variables*, which are identified by names that are *identifiers*. Each variable is associated with a data type and is used to store and refer to values of its type. By default, the value of a variable may change during its lifetime. If the value of a variable never changes after it is first assigned, the variable is known as a *constant* [23]. Declaration of variables and constants are mandatory. The Swift compiler strictly enforces the following rule.

[23] Swift allows you to easily differentiate constants from variables through declarations. It promotes the use of constants whenever possible, since constants offer opportunities for the compiler to optimize the code.

Rule: Declaration and Initialization

1. Every variable or constant must be declared before it can be used.
2. Every variable or constant must be assigned a value before it can be referred to.

Swift uses very similar syntax for declaring variables and constants. You use the keyword `let` to declare a constant and use the keyword `var` to declare a variable.

Syntax: Constant and Variable Declaration

Constant declaration:

`let` *Identifier* : *Type* = *Initial Value*

(*Type*: optional)

Variable declaration:

`var` *Identifier* : *Type* = *Initial Value*

(*Type*: optional; *Initial Value*: optional)

Note that the *Type* for both constant and variable declarations are optional and may be omitted. The *Initial Value* is optional for variable declarations, but required for constant declarations. The *Initial Values* are usually literals, but can be expressions.

PROGRAM 2.1: Constant and Variable Declarations.

Declare a number of constants and variables of various types.

```
let distanceToMoon = 384_400         // km
let earthGravityAcceleration = 9.8   // m/s/s
let languageName = "Swift"
let swiftIsAwesome = true

var total = 100
var velocity = 30.5
var statusMessage = "Success"
var isComplete = false
```

PROGRAM 2.1 OUTPUT:

Results

```
384,400
9.8
"Swift"
true

100
30.5
"Success"
false
```

YOU MAY HAVE NOTICED THAT none of the declarations above includes the type of the constant or variable being declared, which may give the false impression that Swift is dynamically types. To the contrary, Swift is *statically* typed. The following rules are strictly enforced by the Swift compiler.

RULE: STATIC TYPING

1. Every variable or constant is assigned a type at the time of its declaration. The type is either explicitly declared or inferred by the compiler.
2. The type of a variable or constant *cannot* be changed after the declaration.

THE REASON FOR THE TYPES BEING OPTIONAL IN DECLARATIONS is that the Swift compiler is vary capable of inferring the types based on the information available from the context. In the case of constant and variable declarations, if the initial values are provided, the types can be automatically inferred. The rules for inferring types from literal initial values are quite simple as shown in Table 7. Based on these rules, the types of the constants and variables declared in PROGRAM 2.1 are inferred by the compiler as follows:

```
distanceToMoon, total                ⇒  Int
earthGravityAcceleration, velocity   ⇒  Double
languageName, statusMessage          ⇒  String
swiftIsAwesome, isComplete           ⇒  Bool
```

Literals	Inferred Type
Integer literals	`Int`
Floating-point literals	`Double`
String literals	`String`
Boolean literals	`Bool`

Table 7: The default inferred types of literals.

You can always explicitly declare types, as illustrated in Program 2.2. When the declared types are the same as the inferred types, the declared types become redundant.

Program 2.2: Declarations with Explicit Types.

Declare a number of constants and variables with explicit, but redundant types.

```
let distanceToMoon : Int = 384_400           // km
let earthGravityAcceleration : Double = 9.8  // m/s/s
let languageName : String = "Swift"
let swiftIsAwesome : Bool = true

var total : Int = 100
var velocity : Double = 30.5
var statusMessage : String = "Success"
var isComplete : Bool = false
```

Program 2.2 Output:

```
Results
384,400
9.8
"Swift"
true

100
30.5
"Success"
false
```

This program is effectively equivalent to the program in Program 2.1. In this program, the constants and variables are explicitly declared with their types and initial values. Since the declared types are identical to the types inferred from the respective initial values, the declared types are redundant here and can be safely omitted.

Being statically typed means that the type of every constant and variable must be determined at the time of declaration, and cannot be changed thereafter. The type of a declaration must be explicitly declared if the inferred type different from the desired type, or there isn't sufficient information available in the context for the compiler to infer the type. In other words, if no initial value is provided, the type must be explicitly declared [24].

[24] Swift does not provide default initial values to variables or constants of most data types. The only exception is for variables of *optional types*.

PROGRAM 2.3: Declarations with Explicit and Necessary Types.

Declare constants and variables that require explicitly declared types.

```
let currencySymbol : Character = "$"   (1)

let ten : Int32 = 10   (2)
var itemCount : UInt = 0

let length : Float = 100.0   (3)
var totalAmount : Double = 0
```

PROGRAM 2.3 OUTPUT:

```
Results
"$"

10
0

100.0
0.0
```

The constants and variables in this program are all declared with explicit types and initial values. However, the inferred type of each name based on the initial values is different from the declared type. Therefore, the explicitly declared types in this program are significant. Omitting any of the types in this program would result in declarations of constants or variables of different types.

1. Swift does not provide a separate syntax for literals of `Character` type. Character literals are simply string literals with a single character. As far as type inference is concerned, string literals of any length are always inferred as `String` type, even when a string literal contains a single character. To declare a constant or variable of `Character` type, the type must be explicitly declared

2. Integer literals are always inferred as of type `Int`. To declared an integer constant or variable of a different type, the type must be explicitly declared.

3. Similarly, floating-point literals are always inferred as of type `Double`. To declared a floating-point constant or variable of a different type, the type must be explicitly declared

Assignments

Variables declared without an initial value can be assigned values using *assignment statements*.

Syntax: Assignment Statement

Variable = *Expression*

- The left-hand side of an assignment statement must be a single variable.
- The right-hand side can be an arbitrary expression.
- The type of the variable and the type of the expression must be compatible.

In Swift, an assignment is a statement, not an expression. An assignment does not return a value, nor can it be where a value or an expression is expected. So the following code fragments are illegal in Swift

```
a = b = 0
println(a = 10)
```

An assignment statement sets the value of the variable at the left-hand side to the value of the expression at the right-hand side.

Program 2.4: Variable Declarations and Assignments.

Declare variables and assign values separately using assignments.

```
var total : Int
var velocity : Double
var statusMessage : String
var isComplete : Bool

total = 100
velocity = 30.5
statusMessage = "Success"
isComplete = false
```

Program 2.4 Output:

Results

```
100
30.5
"Success"
false
```

In this program, the variables are declared with explicit types, but without initial values. Without the initial values, the complier does not have sufficient information from the context to infer the types of the variables. So, omitting any the of types in the declarations in this program would cause a compile-time error.

Compound Assignments

A *compound assignment* is an abbreviated form of assignment using a compound assignment operator, which is formed by prepending an infix binary operator to the assignment operator (=).

Syntax: Compound Assignment

Variable *op*= *Expression*

The compound assignment above is equivalent to

Variable = (*Variable* *op* *Expression*)

The operator *op* in the compound assignment operator can be any infix binary operator that can operate on the values of the type of the *Expression*. A compound assignment is a statement, not an expression.

Compound assignments	Equivalent to
`i += 1`	`i = i + 1`
`i -= 1`	`i = i - 1`
`d *= 2.0`	`d = d * 2.0`
`d /= 2.0`	`d = d / 2.0`
`str += "Swift"`	`str = str + "Swift"`

Examples of compound assignments and their equivalent forms. Assume `i` is declared as an `Int`, `d` is declared as a `Double`, and `str` is declared as a `String`.

Implicit Assignments

An *implicit assignment* is another abbreviated form of assignment using the increment (++) or decrement (--) operator, which are *unary* operators [25] on variables of numeric types, i.e., integer or floating-point types. The increment and decrement operators can be used as either prefix and postfix operators.

[25] A unary operator operates on a single operand.

Syntax: Implicit Assignments

Prefix increment and decrement:

++ *Variable* -- *Variable*

Postfix increment and decrement:

Variable ++ *Variable* --

The increment operator increments the value of the variable by one, and the decrement operator decrements the value of the variable by one. An implicit assignment is also an expression, which returns a value, in addition to increment or decrement the variable. A prefix operator returns the value of the variable *before* it is incremented or decremented. A postfix operator returns the value of the variable *after* it is incremented or decremented.

Implicit Assignment	Value of the Variable Before	Value of the Implicit Assignment	Value of the Variable After	Same effect as (*w.r.t.* the Variable)
`++i`	10	11	11	`i += 1`
`--i`	10	9	9	`i -= 1`
`i++`	10	10	11	`i += 1`
`i--`	10	10	9	`i -= 1`
`++d`	10.0	11.0	11.0	`d += 1.0`
`--d`	10.0	9.0	9.0	`d -= 1.0`
`d++`	10.0	10.0	11.0	`d += 1.0`
`d--`	10.0	10.0	9.0	`d -= 1.0`

Examples of implicit assignments. Assume `i` is declared as an `Int` and `d` is declared as a `Double`.

Mixed-Type Expressions and Type Conversions

Swift does not allow mixed-type arithmetic expressions, which are expressions whose operands are of different types. Let's consider the following example:

```
let unitPrice = 1.25                    // Type inferred as Double
var quantity = 3                        // Type inferred as Int
var totalPrice = unitPrice * quantity   // A mixed-type expression
```

The mixed-type expression would cause the following compile-time error:

```
cannot invoke '*' with an argument list of type '(Double, Int)'
```

Rule: No Mixed-Type Arithmetic Expressions

1. The operands of an arithmetic expression must be of the same type, and result of the expression is of the same type as the operands.
2. No *implicit* type conversions, i.e., coercions, will be performed, except for literals.

One may consider mixed-type expressions to be quiet natural and convenient. However, to allow mixed-type expressions in a statically typed language would require a set of rules for implicit type conversions, some of which may be considered rather arbitrary. Implicit type conversion is also a potential source of ambiguity and confusion, which may lead to some unexpected behavior of programs or even bugs. Unlike many other languages, Swift adopts the following design rule:

> *Make your intent explicit, and do not reply on implicit rules.*

Operands of an arithmetic expression must be *explicitly* converted to the same type when necessary. Explicit type conversion is expressed using the following syntax:

Syntax: Explicit Type Conversion

Type (*Expression*)

The value of *Expression* is converted to an value of *Type*.

For example:

```
Double(10)    Converts 10 to a Double value 10.0
Int(2.5)      Converts 2.5 to an Int value 2 (by truncating)
```

So the error in the calculation of the `totalPrice` can be fixed in either of the following ways:

```
var totalPrice = unitPrice * Double(quantity)    The result type is Double
```

or

```
var totalPrice = Int(unitPrice) * quantity    The result type is Int
```

THE TYPES OF LITERALS IN SWIFT ARE CONVERTIBLE, i.e. the type of a literal can be *implicitly* converted to the appropriated type expected in the context, in which it appears. An implicit conversion is permissible only if the conversion will not result in any loss of data or precision. For example, the following conversions are permissible:

- Converting an integer literal to `Double` or `Float` type.
- Coverting an integer literal to any integer type, such as `Int8`, or `Int32`, if the value is within the range of the type.
- Converting a non-negative integer literal to any unsigned integer type, such as `UInt` or `UInt16`, if the value is within the range of the type.

On the other hand, converting a floating-point literal to an integer type is not permitted.

The following mixed-type expressions that involve numeric literals are allowed in Swift:

```
5 + 3.14          Integer literal 5 is converted to Double
5.0 + 3           Integer literal 3 is converted to Double
1.25 * 3          Integer literal 3 is converted to Double
unitPrice + 1     Integer literal 1 is converted to Double
var count = 0     Integer literal 0 is converted to UInt
count + 5         Integer literal 5 is converted to UInt
```

The following mixed-type expression is not allowed, since a floating-point literal cannot be converted to `Int`

```
quantity + 2.5
```

String Interpolation

Swift provides a powerful mechanism to build strings by combining string fragments and other values, known as *string interpolation*. An *interpolated* string is string literal that contains one or more *interpolated* expressions that are enclosed between `\(` and `)`. The interpolated expression may not contain an unescaped double quote ("), or an unescaped backslash (\). An interpolated expression can be of any type, and it may refer to any variable or constant in its scope. The value of an interpolated string is the string with each interpolated expression evaluated at run-time and replaced by – including the `\(` and `)` – the string representation of its value.

Program 2.5: String Interpolation

Print the message "*a* time *b* is *a* * *b*" with *a*, *b* and *a* * *b* being replaced by their actual values.

```
let a = 3, b = 5
let result = "\(a) times \(b) is \(a * b)"
```

Program 2.5 Output:

Results

"3 times 5 is 15"

The Control Statements

Control Statements

Swift provide all the control statements that you are familiar with in Java or C++, including

- the `if` statement, and the `if`-`else` statement,
- the `while` loop,
- the `for` loop,
- the `for`-`in` loop, a.k.a., the *for-each* loop and
- the `repeat`-`while` loop,

Additionally, Swift provide a new and very versatile `for`-`in` loop, and a significantly enhanced and considerably more powerful version of the `switch` statement, which we will discuss later.

Conditional Statements

An `if` statement or `if`-`else` statement allows one to choose the path of execution based on a Boolean condition. The meaning of the statements are self-explanatory.

Syntax: Conditional Statements

The `if` statement:

```
if Condition {
    Statements
}
```

The `if`-`else` statement:

```
if Condition {
    Statements
} else {
    Statements
}
```

- The *Condition* is a Boolean expression.
- The *Statements* represent a sequence of zero or more statements.

In Swift, the parentheses around the *Condition* are optional. However, the braces around the *Statements* are mandatory, even when the *Statements* consist of a single statement.

The following is a simple program using a `if-else` statement.

PROGRAM 3.1: Using `if-else` Statement

Prins the message "x / y is x/y" for some x and y when $y \neq 0$. When $y = 0$, print the message "x / y is undefined".

```
var x = 7.5, y = 3.0
if y != 0 {
    print("\(x)/\(y) is \(x/y)")
} else {
    print("\(x)/\(y) is undefined")
}
```

PROGRAM 3.1 OUTPUT:

Results

```
"7.5/3.0 is 2.5"
```

Change the value of y to 0 and see the output of the program.

While Loops

THE SIMPLEST LOOP STATEMENT is the `while` loop, which repeats the *Statements* as long as the *Condition* remains `true`. The syntax of the `while` loop is vary similar to the the `while` loops in Java or C++.

SYNTAX: WHILE LOOP

```
while Condition {
    Statements
}
```

- The *Condition* is a Boolean expression.
- The *Statements* represent a sequence of zero or more statements.

Similar to the conditional statements, the parentheses around the *Condition* are optional. The braces around the *Statements* are mandatory, even when the *Statements* consist of a single statement.

In three of the following programs, PROGRAM 3.2, 3.5, and 3.6, we will use different loop statements to calculate the sums of the following two series:

- The first is a series of integers

$$\sum_{i=1}^{100} i = 1 + 2 + 3 + \cdots + 100 \tag{1}$$

- The second is a series of floating-point numbers

$$\sum_{i=1}^{99} \frac{1}{i} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{99} \tag{2}$$

Program 3.2: Using `while` Loops

Calculate the sums of series (1) and (2).

```
let N = 100
var sum = 0
var i = 1
while i <= N {
    sum += i++
}
print("sum = \(sum)")

var val = 0.0
i = 1
while i < N {
    val += 1.0 / Double(i++)
}
print("val = \(val)")
```

Program 3.2 Output:

```
Output
sum = 5050
val = 5.17737751763962
```

This program calculates the sums of the two series using `while` loops.

Repeat-While Loops

Syntax: Repeat-While Loop

```
repeat {
    Statements
} while Condition
```

- The *Condition* is a Boolean expression.
- The *Statements* represent a sequence of zero or more statements.

Similar to the `while` loop, the parentheses around the *Condition* are optional. The braces around the *Statements* are mandatory, even when the *Statements* consist of a single statement.

The `repeat`-`while` loop is commonly used for loops that are controlled by precision requirements.

The next two programs, *Program* 3.3 and 3.4, illustrate such kind of loops. They calculate the Euler's number[26], e, which is the sum of the following infinite series:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = 1 + \frac{1}{1} + \frac{1}{1 \cdot 2} + \frac{1}{1 \cdot 2 \cdot 3} + \cdots \tag{3}$$

The terms in this infinite series are in a diminishing order. The loop continues until a term is $< 10^{-10}$. The computation can be easily done in either `while` loop or `repeat-while` loop.

[26] The number e, sometimes called Euler's number after Swiss mathematician Leonhard Euler, is an important mathematical constant that is the base of the natural logarithm.

Program 3.3: Using `while` Loop

Calculate Euler's number (3).

```
var n = 1
var t = 1.0, e = 1.0
while (t > 1e-10) {
    var f = 1
    for i in 1 ... n { // calculate n!
        f *= i
    }
    t = 1.0 / Double(f)
    e += t
    n++
}
print(e)
```

Program 3.3 Output:

```
Output
2.71828182845823
```

Program 3.4: Using `repeat-while` Loop

Calculate Euler's number (3).

```
var n = 1
var t = 1.0, e = 1.0
repeat {
    var f = 1
    for i in 1 ... n { // calculate n!
        f *= i
    }
    t = 1.0 / Double(f)
    e += t
    n++
} while (t > 1e-10)
print(e)
```

Program 3.4 Output:

```
Output
2.71828182845823
```

For Loops

Swift also provides a familiar `for` loop with a compact syntax to capture a common idiom of loops: loops that are controlled by variables used as counters. The common parts of this type of loops include: a) initializing a loop control variable; b) using the variable in the condition that controls the loop; and c) incrementing the variable at the end of each iteration.

Syntax: For Loop

```
for Initialization ; Condition ; Increment {
    statements
}
```

(*Initialization*, *Condition*, and *Increment* are optional)

- The *Initialization* is a statement or a declaration for initializing the loop control variable.
- The *Condition* is a Boolean expression.
- The *Increment* is a statement for incrementing or updating the loop control variable.
- The *Statements* represent a sequence of zero or more statements.

It is optional to enclose the part *Initialization* ; *Condition* ; *Increment* in parentheses. The braces around the *Statements* are mandatory, even when the *Statements* consist of a single statement.

The `for` loop is equivalent to the following `while` loop

```
Initialization
while Condition {
    Statements
    Increment
}
```

Program 3.5: Using `for` Loops

Calculate the sums of series (1) and (2).

```
let N = 100
var sum = 0
for var i = 1; i <= N; i++ {
    sum += i
}
print("sum = \(sum)")

var val = 0.0
for var i = 1; i < N; i++ {
    val += 1.0/Double(i);
}
print("val = \(val)")
```

Program 3.5 Output:

```
Output
sum = 5050
val = 5.17737751763962
```

For-In Loops

Swift provides a new form of the `for` loop known as the `for-in` loop.

Syntax: For-In Loop

```
for Variable in Collection {
    Statements
}
```

- The *Variable* is a variable name.
- The *Collection* is an expression whose value must be a collection, i.e., an *array*, a *set*, a *dictionary*, or a *range*.
- The *Statements* represent a sequence of zero or more statements.

It is optional to enclose the part *Variable* `in` *Collection* in parentheses. The braces around the *Statements* are mandatory, even when the *Statements* consist of a single statement.

The `for-in` loop iterates through each value in a collection. Swift provides several types of collections, including *Arrays*, *Dictionaries*, and *Sets*, which we will discuss in detail later. One of simplest way to define collections is to *range expressions*, which define collections of integers.

Syntax: Range Expressions

Closed range expression:

Expression `...` *Expression*

Half-open range expression:

Expression `..<` *Expression*

- Both *Expression* here must expressions of `Int` type.
- The value of the first expression is the lower bound of the range, and the value of the second expression is the upper bound of the range. The values could be positive or negative. The lower bound must be less or equal to the upper bound.
- The *closed range expression* returns a range of integers that starts from the lower bound up to the upper bound, including both the lower and upper bounds.
- The *half-open range expression* returns a range of integers that starts from the lower bound up to the upper bound, including the lower bound but excluding the upper bound.

Examples of range expressions:

Range Expressions	Values in the Ranges
`0 ... 10`	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
`0 ..< 10`	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
`-10 ... 0`	-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0
`-10 ..< 0`	-10, -9, -8, -7, -6, -5, -4, -3, -2, -1
`-1 ... 1`	-1, 0, 1
`1 ... 1`	1
`1 ..< 1`	*(empty range)*

The following range expressions are illegal, because the lower bound of each range is greater than the respective upper bound:

```
1 ..> 0
10 ...  1
-1 ...  -10
```

Program 3.6: Using Range and `for`-`in` Loop

Calculate the sums of series (1) and (2).

```
let N = 100
var sum = 0
for i in 1...N {
    sum += i
}
print("sum = \(sum)")

var val = 0.0
for i in 1..<N {
    val += 1.0/Double(i)
}
print("val = \(val)")
```

Program 3.6 Output:

```
Output
sum = 5050
val = 5.17737751763962
```

Program 3.7: Using Range and `for`-`in` Loop

Print the integer numbers from 0 to 10 in reverse order.

```
for i in -10...0 {
    print("\(-i) ")
}
print()
```

Program 3.7 Output:

```
Output
10 9 8 7 6 5 4 3 2 1 0
```

When the variable in a `for`-`in` loop is not used in the loop's body, the variable can be declared as *anonymous* using the symbol _. This is a common use of the `for`-`in` loop to execute the loop's body a pre-determined number of times.

Program 3.8: Using an Anonymous Variable in a `for`-`in` Loop

Calculate the value of 2^{10}

```
var powerOfTwo = 1
for _ in 1 ... 10 {
    powerOfTwo *= 2
}
print(powerOfTwo)
```

Program 3.8 Output:

```
Output
1024
```

This program calculates the value of 2^{10} by simply executing the loop's body 10 times.

Using Arrays

A simple and useful collection type in Swift is the `Array` type. In Swift, an *array* refers to an ordered collection of values of a uniform type. Unlike arrays in C++ or Java, arrays in Swift can have flexible sizes[27].

[27] Stay tuned for a full discussion of Swift collection types, which include `Array` and `Dictionary`.

A constant array is defined as an *array literal*, using the following syntax.

Syntax: Array Literal

[*Expression*$_1$, *Expression*$_2$, ...]

(*Expression*$_2$, ... : optional)

- All *Expressions* must be of the same type.
- The *Expressions* are optional. When no expression is present, it represents an empty array.

Given a constant or variable *a* of an array type, the following simple operations on arrays are supported

a`.count`	returns the size of the array
a`[ i ]`	returns the value of the *i*-th element in the array

The following is a simple program illustrating the use of array literals and basic operations.

Program 3.9: Linear Search

Determine if a given city is among the 25 largest cities by population in Europe.

```
let largest25EuropeanCities = [
    "London", "Berlin", "Madrid", "Rome", "Paris",
    "Bucharest", "Vienna", "Budapest",  "Hamburg",
    "Warsaw", "Barcelona", "Munich", "Milan",
    "Prague", "Sofia", "Brussels", "Birmingham",
    "Cologne", "Naples", "Stockholm", "Turin",
    "Marseille", "Amsterdam", "Valencia", "Zagreb"
]
let searches = [
    "Rome", "Venice", "Barcelona", "Seville"
]
for city in searches {
    var found = false
    for c in largest25EuropeanCities {
        if c == city {
            found = true
            break
        }
    }
    print(city, "is", found ? "found" : "not found")
}
```

Program 3.9 Output:

Output

```
Rome is found
Venice is not found
Barcelona is found
Seville is not found
```

This program declares two array constants. The first array, `largest25EuropeanCities`, is an array of the names of the 25 largest cities by population in Europe. The second array, `searches`, is an array of city names we want to determine if they are among the 25 largest cities in Europe. The program performs a linear search of each of the names in `searches`, and prints out a message of whether it is found in the array of the 25 largest cities in Europe or not.

Functions

Functions

A function is a self-contained block of code that performs a certain task. The function can be called to perform the task when needed. In Swift, functions are declared in the global scope. They do not have to belong to some class, and they are accessible everywhere in the program.

Function Declarations and Calls

A *function declaration* begins with the `func` keyword, followed by a *name*, and an optional list of *parameters*.

Syntax: Function Declaration

```
func Identifier ( Parameter1 , Parameter2 , ... )      (optional)
        -> Type                                        (optional)
    Statements
}
```

- The *Identifier* is the name of the function.
- The *Parameters* are optional.
- The *Type* refers to the type of the return value of the function, and it is optional.
- The *Statements* is the body of the function.

For a function that does not return a value, the return type can be simply omitted. You do not declare the return type to be `void` in Swift.

A *function call* begins with the function name, followed by a list of optional *arguments*. In Swift, the arguments in function calls can be *anonymous* or *named* The *function calls with anonymous arguments* has a similar syntax to function calls in Java and C.

Syntax: Function Call with Anonymous Arguments

```
Identifier ( Expression1 , Expression2 , ... )      (optional)
```

- The *Identifier* is the name of the function.
- The *Expression* are the optional arguments of the function call.

The following are several examples of simple function declarations and calls.

Program 4.1: A Parameter-less Function

Define a function that prints the message "Hello, world!".

Program 4.1 Output:

```
Output
Hello, world!
```

The `helloWorld` function takes no parameters and returns no value.

Parameters

A function may take one or more parameters. A parameter can be declared as follows:

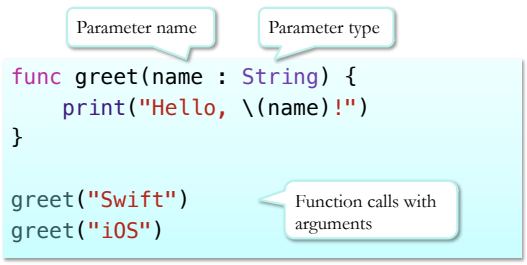
Syntax: Function Parameter

Identifier : *Type*

- The *Identifier* is the name of the parameter.
- The *Type* is the type of the parameter. The type of each parameter must be explicitly declared.

Program 4.2: A Function with a Parameter

Define a function that prints the message "Hello, *name*!", where *name* is a parameter of the function.



Program 4.2 Output:

```
Output
Hello, Swift!
Hello, iOS 8!
```

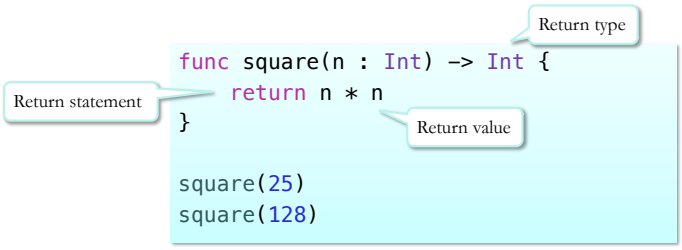
The `greet` function takes one parameter and returns no value.

Return Values

Functions may return values. If a function returns a value, the return type must be declared. If no return type is specified, the function does not return a value. If a function returns a value, then a `return` statment with a value must be present in every path of the body if the function.

Program 4.3: A Function with a Return Value

Define a function that takes an integer parameter and returns the square of the input parameter.



Program 4.3 Output:

Results

625
16,384

Program 4.4: Primality Test

Define a Boolean function with an integer parameter that determines whether the parameter is a prime number.

```
func isPrime(n : Int) -> Bool {
    if n < 2 {
        return false
    } else {
        for var i = 2; i * i < n; i++ {
            if n % i == 0 {
                return false
            }
        }
        return true
    }
}

isPrime(2)
isPrime(15)
isPrime(23)
```

Program 4.4 Output:

Results

true
false
true

Program 4.5: Convert to Ordinals

Define a function with an integer parameter that converts the input parameter to its corresponding ordinal number by appending one of the suffixes: "st", "nd", "rd", or "th", according to the English convention, i.e., converting 1, 2, 3, 4, ... to 1st, 2nd, 3rd, 4th, ..., etc.

```
func toOrdinal(i: UInt) -> String {
    if i % 10 >= 1 && i % 10 <= 3  // 1st, 2nd, 3rd
        && i % 100 / 10 != 1 {     // but 11th, 12th, 13th
        let suffix = [ "st", "nd", "rd" ]
        return "\(i)\(suffix[i % 10 - 1])"
    } else {
        return "\(i)th"
    }
}

toOrdinal(1)
toOrdinal(2)
toOrdinal(3)
toOrdinal(4)
toOrdinal(10)
toOrdinal(11)
toOrdinal(15)
toOrdinal(23)
```

Program 4.5 Output:

Results

```
1st
2nd
3rd
4th
10th
11th
15th
23rd
```

Program 4.6: A Recursive Function: the *Fibonacci* Numbers

Define a function to compute the i-th *Fibonacci* number, where i is the input parameter. The *Fibonacci* numbers are defined as

$$Fibonacci(i) = \begin{cases} 1 & i = 0, 1 \\ Fibonacci(i-1) + Fibonacci(i-2) & i > 1 \end{cases}$$

```
func fibonacci(i : UInt) -> UInt {
    if (i == 0 || i == 1) {
        return 1
    } else {
        return fibonacci(i - 1) + fibonacci(i - 2)
    }
}

fibonacci(8)
fibonacci(12)
```

Program 4.6 Output:

Results

```
34
233
```

Multiple Parameters

A function may have multiple parameters. Let's consider the following example of a simple function maximum with two parameters.

Program 4.7: The Maximum Function

Define a function that returns the maximum of the two input parameters, which are integers.

```
func maximum(x: Int, y: Int) -> Int {
    return x >= y ? x : y
}
```

If we call the function as follows

```
maximum(2, 5)
```

It would cause a compile error. This is because Swift adopts a different style of syntax for function parameters, i.e., *named parameters*. The default for the first parameter is anonymous, but the default for the second and subsequent parameters is named.

Swift allows parameters to be named, so that the programs can be more readable. Let's revisit the syntax for function parameters in function declarations. Swift allows each parameter to have two names, an optional *external name* in addition to the required *local name*.

Syntax: Function Parameter

Function parameter with an external name:

Identifier (optional) *Identifier* : *Type*

- The first *Identifier* is the *external name* of the parameter. A special value of the external name is _, which indicates an anonymous external name. The external name is optional.
- The second *Identifier* is the *local name* of the parameter.

Syntax: Function Call with Named Arguments

Identifier (*Identifier*$_1$ (optional) : *Expression*$_1$, *Identifier*$_2$ (optional) : *Expression*$_2$, ...)

- The first *Identifier* is the name of the function.
- Each *Identifier*$_i$ that precedes an expression is the *external name* of the corresponding parameter at the i-position.
- Each *Expression*$_i$ is the argument for corresponding parameter at the i-position.

Important: Even when the arguments are named, the arguments in function calls must appear in the same order as they are in the function declarations.

The local names are used in the function body, and the external names are used as the *argument names* in function calls. If no external name is specified, the default external names of parameters are determined based on the following simple rules.

Rule: Default External Parameter Name

For a parameter in a function declaration,

- the default external parameter name of the *first* parameter is *anonymous*, i.e., _
- the default external parameter name of the second and all subsequent parameters is the internal parameter name.

So the proper way to call the maximum function declared is as follows:

```
maximum(2, y: 5)
```

Note that the external name of the second parameter is y, the local name, by default.

This function call works, but seemed unnatural. It treated two symmetrical parameters x and y in an asymmetrical way. The question is: what is the purpose of the external names of parameter? Answer: to improve the readability of the function calls. Consider following alternative declaration of the maximum function

The Maximum Function, Version 2

```
func maximumOf(x: Int, and y: Int) -> Int {
    return x >= y ? x : y
}
```

Note the "Of" at the end of the name, and the explicit external name "and" of the second parameter. Now the function can be call as follows:

```
maximumOf(2, and: 5)
```

It makes the function call more readable. It reads like a sentence:

(the) maximum of 2 and 5 ...

Using External Parameter Names

Have you seen cryptic function calls like this?

```
printTicket("Paris", "Boston", "Orlando")
```

What are the meanings of the arguments? Are they names of places or person? Which one is the name of a person and which one are the names of places? Perhaps, the arguments of the following call contains some clue,

```
printTicket("Tim Cook",
            "San Francisco, CA",
            "Chicago, IL")
```

But, is Tim Cook coming to Chicago, or is he leaving from Chicago? Can you figure out the meanings without searching for the declaration of the function?

Let's see how external names can help improve the readability of your code. Consider the following program with several alternative versions of the printTicket function.

PROGRAM 4.8: Functions with Named Parameters

Define a function to print a simplified airplane ticket with the passenger name, the origin and the destination of the flight.

Version 1

```
func printTicket(name: String, origin: String,
                 destination: String) {
    print("Ticket\n  Passenger name: \(name)")
    print("  From: \(origin)\n  To: \(destination)")
}

printTicket("Tim Cook",
            origin: "San Francisco, CA",
            destination: "Chicago, IL")
```

PROGRAM 4.8 OUTPUT:

Output

```
Ticket
  Passenger name: Tim Cook
  From: San Francisco, CA
  To: Chicago, IL
```

This version of the function declaration is rather simple and straightforward. The function declaration uses the default external names for the 2nd and 3rd parameters. The function call with the external names makes the meanings of the argument clear and unambiguous. But the function call does not read quite like a fluent sentence.

TO MAKE FUNCTION CALLS CLEARER AND MORE READABLE, each argument in a function call may be proceeded by a name. In function declarations, each parameter may have optional *external name* in addition to the required *local name*.

The local name is used in the function body to refer to the parameter. The local name of a parameter is usually chosen to express the intent of the parameter, such as origin and destination. The external name of a parameter is used in the function call, placed before the corresponding argument. Carefully chosen external names can improve the readability of the function call.

Version 2

```
func printTicketFor(name: String, from origin: String,
                    to destination: String) {
    print("Ticket\n  Passenger name: \(name)")
    print("  From: \(origin)\n  To: \(destination)")
}

printTicketFor("Tim Cook",
               from: "San Francisco, CA",
               to: "Chicago, IL")
```

In this version, we try to improve the readability of the function call. We add a connective word *for* to the function name, and choose to use external names for the second and third parameter, the origin and destination, respectively. Now, with the help of the connective word and the external name, the function call of `printTicketFor` is much more readable on its own. It reads like a proper sentence:

print ticket for *Tim Cook,* from *San Francisco, CA,* to *Chicago, IL.*

Okay, if you fall in love with this new style advocated by Swift, great! But if you are traditionalist and favor brevity whenever the meanings are clear from the context, do not despair. Swift allows an easy way to revert to the traditional style of function calls with anonymous arguments. You can simply use _ to indicate an anonymous external name. This is the default for the first parameter of functions. If you want the other parameters to be anonymous, you must explicit indicate that with _.

Program 4.9: The Median Function

Define a function that returns the median of the three input parameters, which are integers.

```
func median(x: Int, _ y: Int, _ z: Int) -> Int {
    return x > y ? (y > z ? y : x > z ? z : x)
        : (x > z ? x : y > z ? z : y)
}

median(1, 2, 3)
median(3, 1, 2)
```

Program 4.9 Output:

Results

```
2
2
```

Or you can do this in the Swift way:

```
func medianOf(x: Int, and y: Int, and z: Int) -> Int {
    return x > y ? (y > z ? y : x > z ? z : x)
        : (x > z ? x : y > z ? z : y)
}

medianOf(1, and : 2, and: 3)
```

Again, the function call reads more like a sentence

(the) median of *2* and *5* and *3*

It is simply a matter of style.

Program 4.10: The Greatest Common Divisor Function

Define a function that computes the *greatest common divisor (gcd)* of the two input parameters, which are positive integers, using Euclid's algorithm.

```
func gcd(a: UInt, _ b: UInt) -> UInt {
    var u = a, v = b
    var r = u % v
    while r > 0 {
        u = v; v = r; r = u % v
    }
    return v
}

gcd(2, 3)
gcd(5, 15)
gcd(18, 15)
gcd(18, 27)
```

Program 4.10 Output:

Results

1
5
3
9

With anonymous parameters.

```
func gcd(a a: UInt, b: UInt) -> UInt {
    var u = a, v = b
    var r = u % v
    while r > 0 {
        u = v; v = r; r = u % v
    }
    return v
}

gcd(a: 2, b: 3)
gcd(a: 5, b: 15)
gcd(a: 18, b: 15)
gcd(a: 18, b: 27)
```

With named parameters.

Default Parameter Values

In Swift, you can provide a default value for a parameter. The default value is used when an argument is not provided in a function call for this parameter.

Syntax: Function Parameter

Function parameter with a default value:

Identifier *Identifier* : *Type* = *Expression*

(*Identifier* optional; = *Expression* optional)

- The first *Identifier* is the *external name* of the parameter.
- The second *Identifier* is the *local name* of the parameter.
- The *Expression* is the default value for the parameter. When the default value is present, an external name should be explicitly specified, or it is assumed to be the same as the local name by default.

When default values are provided to some parameters in a function declaration, it may introduce some ambiguity and/or confusion when matching arguments to parameters. To avoid such ambiguity and/or confusion, parameters with default values should have explicit external names. It is a good practice to have external names for all parameters when default values are provided for some parameters.

Program 4.11: A Function with a Default Parameter Value

Define a function that prints the message "Hello, *name*!", where *name* is a parameter of the function. When no name is provided in a call, it prints "Hello, world!"

```
func greeting(name : String = "world") {
    print("Hello, \(name)!")
}

greeting("Swift")
greeting()
```

Program 4.11 Output:

```
Output
Hello, Swift!
Hello, world!
```

To avoid ambiguity, parameters with default values and anonymous external name should only appear at the end of the parameter list.

Classes

Classes and Objects

A *class* REPRESENTS A GROUP OF *objects*, known as its instances, that share some common characteristics and/or behaviors. Each class defines a *type*, to which its instances belong.

Class Declaration

A CLASS IS DEFINED AS a set of *properties* and *methods* that are shared by all its instances. Properties are attributes or values of the objects. Methods are functions or tasks that can be performed by the objects. A class declaration include[28] :

- A class name.
- Its superclass and the adopted protocols.
- Declarations of class members, including properties, methods, and initializers.

[28] Swift does not separate class interfaces from implementations A class declaration is contained in a single file.

SYNTAX: CLASS DECLARATION

```
class Identifier : Superclass (optional) , Protocols (optional) {
    Member Declarations
}
```

- The *Identifier* is the name of the class.
- The *Superclass* is the name of the superclass.
- The *Protocols* is a list of names of the protocols adopted by the class.
- The *Member Declarations* is a list of declarations that include the declarations of properties, methods, initializers, and other class members.
- *Properties* are declared using the same syntax of constant and variable declarations.
- *Methods* are declared using the same syntax of function declarations.

The topics of inheriting from a superclass and adopting protocols will be discussed later.

Properties and methods of a class are declared using the same syntax for declaring constants, variables, and functions in the global scope.

A class property can be declared as a variable, which is mutable, or a constant, which is immutable. A property declared as a constant or variable is known as a *stored property*[29], which means the value of the property is stored on every instance of the class.

[29] Swift also supports *computed properties*, which will be discussed later.

Creating & Using Instances

Creating an instance of a class consists of two steps:

1. Allocation of the memory for the instance;
2. Initialization of the properties of the instance.

That task of allocation is automatically handled by the system, and the task of initialization is handled by one of the *initializers* in the class declaration. The initializers are responsible for setting the initial values of all the properties of a new instance.

Instances of a class can be created by calling one of the initializers of the class to perform the initialization of the instance . The syntax for calling an initializer of a class is similar to a function call, with the class name being used as the function name.

It is not necessary to explicitly call `alloc` for allocation in Swift. When creating a new instance, it is implied that the allocation of the memory always precedes the initialization.

Syntax: Create an Instance of a Class

Create an instance initialized using the default initializer:

Identifier ( )

Create an instance initialized using a user-defined initializer:

Identifier (*Arguments*) — *Arguments*: optional

- The *Identifier* is the name of the class, of which an instance will be created.
- The expression returns a new instance of the class initialized by the initializer of the class that matches the arguments provided.

The *default initializer* of a class takes no argument. If no initializer is declared, Swift automatically provides a default initializer, which initializes all the properties of a new instance to their respective initial values specified in the property declarations. User-defined initializers will be discussed in *Initializers* (p. 53)

To access the properties and methods of an object, you can simply use the *dot* syntax.

Syntax: Access Class Members

Read the value of a property:

Object . *Property*

Set a value to a property:

Object . *Property* = *Expression*

Call a method:

Object . *Method* (*Arguments*) — *Arguments* optional

- The *Object* is an expression that evaluates to an object.
- The *Property* is the name of the property to be accessed.
- The *Method* is the name of the method to be called.

Let's look at some simple examples of classes.

Program 5.1: The Counter Class: the Initial Version

Define a Counter class that models a simple integer counter. The class will have an integer property count and a method increment, which increments the count property.

```
class Counter {
    var count = 0
    func increment() {
        count++
    }
}
```

Note that no initializer is declared in the Counter class. So the default initializer is provided implicitly. We can create an instance of the Counter class and access its property and method. The instance is initialized using the default initializer.

```
var c1 = Counter()
c1.increment()
c1.increment()
c1.count
c1.count = 0
c1.increment()
c1.count
```

Program 5.1 Output:

Results

```
Counter
Counter
Counter
2
Counter
Counter
1
```

Program 5.2: The Counter Class, Version 2

Add the following methods to the Counter class:

- A decrement method, which decrements the count property.
- A incrementBy and a decrementBy method, which respectively increment and decrement the count property by an amount specified by a parameter.

```
class Counter {
    var count = 0
    func increment() {
        count++
    }
    func decrement() {
        count--
    }
    func incrementBy(c: Int) {
        count += c
    }
    func decrementBy(c: Int) {
        count -= c
    }
}
```

An instance of the new class and some operations on the instance.

```
var c2 = Counter()
c2.incrementBy(10)
c2.count
c2.decrement()
c2.decrementBy(5)
c2.count
```

Program 5.2 Output:

Results

```
Counter
Counter
10
Counter
Counter
4
```

PROGRAM 5.3: The Fraction Class, the Initial Version

Define a class Fraction that represents a fraction number a/b, where both a and b are integers and the denominator $b > 0$. Provide the following methods:

- The printFraction method, which prints the fraction to the standard output in the form of a/b
- The toDouble method, which converts the fraction to it equivalent value in `Double`.

```
class Fraction {
    var numerator: Int = 0
    var denominator: Int = 1

    func printFraction() {
        print("\(numerator)/\(denominator)")
    }

    func toDouble() -> Double {
        return Double(numerator) / Double(denominator);
    }
}
```

This initial version of Fraction does not provide an initializer. The default initializer initializes new instances to $0/1$, the initial values of the numerator and denominator, respectively.

```
var f1 = Fraction()
f1.printFraction()
f1.numerator = 1
f1.denominator = 3
f1.printFraction()
print(f1.numerator)
print(f1.denominator)
print(f1.toDouble())
```

PROGRAM 5.3 OUTPUT:

Results

```
Fraction
Fraction
Fraction
Fraction
Fraction
Fraction
"1\n"
"3\n"
"0.333333333333333\n"
```

Initializers

A class may declare one or more Initializers to initialize the state of a new instance. Initializer declarations are similar to function declarations.

Syntax: Initializer Declaration

```
init ( Parameter1 , Parameter2 , ... ) {
    Statements
}
```

(Parameter2 and the trailing , ... are optional)

- The name of the initializer must be `init`.
- An initializer may take optional parameters.
- No return type is allowed.
- The *Statements* is the body of the initializer. It should not return any value.

One of the requirements of class initialization is the following:

Rule: Class Initialization

Every property must be assigned an initial value either in its declaration or in *every* initializer.

Declarations of functions in the global scope and the initializers and methods inside a class generally follow the same rules, with one exception – the default external name of the *first* parameter of an initializer.

Rule: Default External Parameter Name

For the parameters of a class initializer, the default external name is always the same as the local name, including the first parameter.

In other word, if you want the first parameter of an initializer to be anonymous, you must explicitly declare the external name to be `_`.

PROGRAM 5.4: The Fraction Class, Version 2

Enhance the initial version of the Fraction class by

- Defining initializers.
- Adding methods to set both the numerator and denominator.

```
class Fraction {
    var numerator: Int = 0
    var denominator: Int = 1

    init(numerator: Int, denominator: Int) {           1
        self.numerator = numerator
        self.denominator = denominator
    }

    init(_ numerator: Int, over denominator: Int) {    2
        self.numerator = numerator
        self.denominator = denominator
    }

    init() {}                                          3

    func setTo(numerator: Int, denominator: Int) {     4
        self.numerator = numerator
        self.denominator = denominator
    }

    func setTo(numerator: Int, over denominator: Int) { 5
        self.numerator = numerator
        self.denominator = denominator
    }

    func setTo(numerator: Int, _ denominator: Int) {   6
        self.numerator = numerator
        self.denominator = denominator
    }

    func printFraction() {
        print("\(numerator)/\(denominator)")
    }

    func toDouble() -> Double {
        return Double(numerator) / Double(denominator);
    }
}
```

In this version of the Fraction class, three initializer are defined. The first initializer (1) takes two parameters for the initial values of the numerator and denominator, respectively. Notice the use of the keyword `self` in the initializer body [30]. The keyword `self` is commonly used in the initializers and methods of classes to refer to the object *itself*, i.e., the object on which the initialization or operation is being performed. The use of the keyword `self` in the initializer illustrates a common idiom of using `self` – to differentiate a property of class from a parameter of the initializer, both of which use the same name.

[30] The keyword `self` in Swift is equivalent to the keyword `this` in Java and C++.

In the first initializer (1), neither parameter has an external name. For initializers, the default external parameter name is the interna name. To use the first initializer to initialize an instance of the Fraction class, external names must be used for the arguments of the initializer call.

```
var f1 = Fraction(numerator: 1, denominator: 2)
f1.printFraction()
```

To make the initializer call more succinct and readable, we define a second initializer (2) with explicitly declared external parameter names. The following is a call to the second initializer.

```
var f2 = Fraction(2, over: 3)
f2.printFraction()
```

Since we have explicitly declared initializers in this class, the default initializer is no longer implicitly provided. If we still want to use the default initializer, it must be explicitly declared as well. We have declared a default initializer (3) with an empty body, which means it simply uses the initial values in the declaration to initialize the properties[31].

[31] This is the same behavior as the implicitly provided default initializer.

We also define several different versions of the method setTo to set the numerator and denominator of the fraction. The first version of the method (4) takes two parameters for the respective values of numerator and denominator, and neither of them has an external name. Note that how the first and the rest of the parameters in a method declaration are treated differently with respect to the default external parameter names.

The following is an instance of the Fraction class initialized using the default individualizer, and then use the setTo method to set the values of the numerator and denominator. The external name of the first parameter of setTo defaults to anonymous, and the external name of the second parameter defaults to the local name.

```
var f3 = Fraction()
f3.setTo(1, denominator: 3)
f3.printFraction()
```

To make the setTo method call more succinct and readable, we define a second version () of the method with explicitly declared external parameter names. The following is a call to the second version of the method.

```
var f4 = Fraction()
f4.setTo(1, over: 4)
f4.printFraction()
```

The third version () of the setTo method explicitly declare the external parameter to anonymous.

```
var f5 = Fraction()
f5.setTo(3, 4)
f5.printFraction()
```

Overloading

You may have noticed in the previous program that we have defined several methods of the same name, and we have also defined several initializers (of the same name). This is known as *overloading*. Swift allows overloading of functions, methods, and initializers, provided that calls to each overloaded function, method, or initializer are differentiable from calls to others.

Rule: Overloading

Functions, methods, and initializers can be overloaded. They must be differentiable in one or both of the following aspects

- They take different number of parameters.
- At least on of the parameters at a given position has different types, or different *external* names.

Differences in the *local* name of a parameter alone is not sufficient to differentiate two overloaded functions, methods, or initializers.

PROGRAM 5.5: The Fraction Class, Version 3

Enhance the Fraction class to support the *add* operation of two fraction numbers. The addition of two fraction numbers can be computed using the following formular:

$$\frac{a}{b} + \frac{c}{d} = \frac{a * d + b * c}{b * d}$$

```
class Fraction {
    var numerator: Int = 0
    var denominator: Int = 1

    Initializer and method declarations

    func add(f: Fraction) {
        numerator = numerator * f.denominator +
                    denominator * f.numerator;
        denominator = denominator * f.denominator;
    }

    func reduce() {
        let sign = numerator >= 0 ? 1 : -1
        var u = numerator * sign
        var v = denominator
        var r: Int
        while (v != 0) {
            r = u % v; u = v; v = r
        }
        numerator /= u * sign;
        denominator /= u;
    }

}
```

The add method is a straightforward implementation of the addition formular. However, the result fraction may not be in the simplest form, i,e., there are common divisors of the numerator and denominator that are > 1. For example, $1/2 + 1/4$ yields $6/8$. The reduce method is to simplify the results using the Euclid algorithm to find the greatest common divisor between the the numerator and denominator, and divide the numerator and denominator by their greatest common divisor.

```
var f1 = Fraction(1, over: 2)
var f2 = Fraction(1, over: 4)

f1.add(f2)
f1.printFraction()

f1.reduce()
f1.printFraction()
```

Program 5.5 Output:

```
Output
6/8
3/4
```

Now we have the result we want, but we have to explicitly call the reduce method to get the right result. We'd like to get the right result without this extra step.

Program 5.6: The Fraction Class, Version 4

Improve the Fraction class. First, we modify the add method so that it directly yields a result that is in its simplest form.

```
class Fraction {
    …
    func add(f: Fraction) {
        numerator = numerator * f.denominator
                    + denominator * f.numerator;
        denominator = denominator * f.denominator;
        reduce()
    }
    …
}
```

This version of the add method calls the reduce method at the end. So that it directly yields a result that is in its simplest form.

```
var f1 = Fraction(1, over: 2)
var f2 = Fraction(1, over: 4)
f1.add(f2)
f1.printFraction()
```

Program 5.6 Output:

```
Output
3/4
```

Next, we define another add method that returns the result as a new fraction object.

```
class Fraction {
    …
    func add(f: Fraction) -> Fraction {
        var result: Fraction = Fraction()
        result.numerator = numerator * f.denominator
           + denominator * f.numerator;
        result.denominator = denominator * f.denominator;
        result.reduce()
        return result
    }
    …
}
```

The following is a simple example of using this version of the add method. After the addition, the result is in a new Fraction object f3. The orignial Fraction objects f1 abd f2 retain their respective states.

```
var f1 = Fraction(1, over: 2)
var f2 = Fraction(1, over: 4)
let f3 = f1.add(f2)
f3.printFraction()
```

Program 5.6 Output:

Output

```
3/4
```

We can take this Fraction class example one step further.

PROGRAM 5.7: The Fraction Functions and Operators

Define the add operation on two fraction numbers as a global function. Also allow operators + and += to operate on Fraction objects.

```
class Fraction {
    …
    func add(f: Fraction) -> Fraction { … }
    …
}

func add(a: Fraction, b: Fraction) -> Fraction {    (1)
    return a.add(b)
}

func +(a: Fraction, b: Fraction) -> Fraction {      (2)
    return a.add(b)
}

func +=(inout left: Fraction, right: Fraction) {    (3)
    left = left + right
}
```

The global function add (1) allows the addition of two Fraction object to be performed as a function call, rather than a method call on an object. The two operands of the addition are treated symmetrically this way.

```
var f1 = Fraction(1, over: 2)
var f2 = Fraction(1, over: 4)

let f4 = add(f1, f2)
```

Swift supports *operator overloading*, which allows operators to operate on any user defined types. This is accomplished by simply allowing operators to be used as names of functions defined in the global scope. We defined a global function for operator + that takes parameters of Fraction objects (2). This function enables operator + to operate on Fraction objects.

```
let f5 = f1 + f2
```

Furthermore, we can also overload the operator += by providing a function declaration (3). Now, we can do:

```
var f6 = Fraction(1, over: 2)
f6 += f2
```

Access Control

Swift provides three different access levels[32]:

- *Public* entities are accessible anywhere within your project and available to other projects. You typically use public access when specifying the public interface of a reusable module or framework.
- *Internal* entities are only accessible anywhere within the *module*, in which they are defined. This is the default access level. Typically, a simple project would consists of a single module. In this case, internal entities are accessible anywhere within your project. Use internal access for entities that are designed for a your project only.
- *Private* entities are only accesible within the source file, in which they are defined. Use private access to hide the implementation details.

You may place one of the access level keywords: `public`, `internal`, or `private` in front of declarations of constants, variables, classes, etc, to indicate the desired access level of the declarations.

[32] You may have noticed that Swift does not provide an inheritance based access level similar to the *protected* access in Java and other languages. This has been the source of some controversy.

We will discuss class inheritance, extension, protocols, and other advanced feature of classes in a separate lecture.