FITPED

iOS - Swift fundamentals

Dalibor Kunhart
Peter Švec
Kristián Fodor

www.fitped.eu

2021

Co-funded by the
Erasmus+ Programme
of the European Union

Work-Based Learning in Future
IT Professionals Education
(Grant. no. 2018-1-SK01-KA203-046382)

iOS – Swift Fundamentals

Published on

November 2021

Authors

Dalibor Kunhart | Mendel University in Brno, Czech Republic

Peter Švec | Teacher.sk, Slovakia

Kristián Fodor | Constantine the Philosopher University in Nitra, Slovakia

Reviewers

Cyril Klimeš | Mendel University in Brno, Czech Republic

Ján Skalka | Constantine the Philosopher University in Nitra, Slovakia

Eugenia Smyrnova-Trybulska | University of Silesia in Katowice, Poland

Piet Kommers | Helix5, Netherland

Graphics

Ľubomír Benko | Constantine the Philosopher University in Nitra, Slovakia

David Sabol | Constantine the Philosopher University in Nitra, Slovakia

Erasmus+ FITPED

Work-Based Learning in Future IT Professionals Education

Project 2018-1-SK01-KA203-046382

Co-funded by the
Erasmus+ Programme
of the European Union

ISBN 978-80-558-1791-0

Table of Contents

Motivational Introduction

Chapter 1

1.1 Introduction

1.1.1

Swift is a new programming language for running applications on iOS and macOS operating systems. Swift builds on the best of C and Objective-C. It is adopting secure programming patterns and add advanced features that make programming easier, more flexible and more fun.

1.1.2

Which manufacturer develops and operates iOS hardware?

- Microsoft
- Apple
- Google

1.1.3

Swift simplifies memory management with automatic reference counting (ARC). It is nice for Objective-C developers. Accepts the readability of named parameters and the power of the Objective-C dynamic object model. It supports Playgrounds, an innovative feature that allows programmers to experiment with Swift code and see the results instantly, without the hassle of compiling and running the application. Thanks to all this, Swift is a new language for the future of software development for Apple.

1.1.4

What languages is Swift built on?

- Pascal
- C and Objective-C
- Unix

1.1.5

Tradition dictates that the first program in the new language should display "Hello, world!". In Swift, this can be done in one line:

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

1.1.6

Write the following greeting in Swift: "Hello, I am here!"

1.1.7

If you have programmed in C or Objective-C before, this syntax seems familiar to you - in Swift, this line of code is a complete program. You do not need to import a separate library for functions such as input/output or string processing. Your main code is used as the entry point to the program, so you do not need the main function. You do not have to write a semicolon at the end of the line.

1.1.8

Do I need to import an input / output library in Swift?

- no
- yes

1.2 Casting, Arrays, Dictionary

1.2.1

Casting 1

Values are never implicitly converted to another type. If you need to convert a value to another type, explicitly create an instance of the desired type.

```
let label = "Width is "
let width = 94
let widthLabel = label + string (width)
```

1.2.2

Arrays are created using square brackets [] and their elements are accessed by writing the index in parentheses.

```
var colors = ["white", "blue", "red", "green"]
colors [1] = "yellow"
```

To create an empty array, use the following syntax:

```
let emptyArray = [string] ()
```

1.2.3

If type information can be derived, you can write an empty field as [] - for example, when assigning a new value to a variable or passing a value to a function.

```
array = []
```

1.2.4

For example, you create a dictionary using square brackets [] and access their elements by writing the key in parentheses.

```
var flowerColors = ["daisy": "white", "dandelion": "yellow"]
flowerColors ["tulip"] = "red"
```

and to create an empty dictionary, use the following syntax:

```
let emptyDictionary = [String: Float] ()
```

If type information can be derived, you can write an empty dictionary as [:] - for example, when assigning a new value to a variable or passing a value to a function.

```
dictionary = [:]
```

1.2.5

How do you explicitly convert 4.5 to a string?

- string(4.5)
- str(4.5)
- string to (4.5)

1.2.6

Casting 2

There is an even easier way to include values in strings: Write the value in parentheses and precede the parentheses with a backslash \(...).

```
let apples = 3
let oranges = 5
let Fruits = "I have \(apples) apples and \(oranges) oranges.
Together I have \(apples + oranges) pieces of fruits."
```

1.2.7

How do you add the first and the last name of the user to the following sentence "User…… is currently logged in"

- let message = "User \(firstname) \(lastname) is currently logged in"
- let message = "User + (firstname) + (lastname) is currently logged in"
- let message = "User \(firstname)\(lastname) is currently logged in"

1.2.8

Select the correct array notation for real numbers 10, 15, 18, and 19

- var arr = [10.0, 15, 18, 19]
- var arr [10.0, 15, 18, 19]
- var arr is [10.0, 15, 18, 19]

1.2.9

Create a dictionary called *position* with real values x = 10, y = 20

Control Flow

Chapter 2

2.1 Loops

📖 2.1.1

Control Flow

Swift provides a number of control flow commands. These include while loops to perform a task multiple times; if, guard, and switch statements to execute different branches of code based on certain conditions; and statements like break or continue to move to another location in your code.

Swift also has a for-in loop that makes it easy to iterate over an array, dictionary, set, string, and other types.

📖 2.1.2

For-in loops

Use a for-in loop to iterate a sequence, such as items in an array, numeric ranges, or characters in a string.

📖 2.1.3

for-in over an array

This example uses a for-in loop to iterate over an array:

```
let names = ["Anna", "Alex", "Brian", "Jack"]
for name in names {
  print ("Hello, \(name)!")
}
// Hello, Anna!
// Hello, Alex!
// Hello, Brian!
// Hello, Jack!
```

2.1.4

What will be the data type of the variable *a* in the loop?

```
for a in [1, 4, 5.6, 7, 8.9]
```

- Double
- Int
- String

2.1.5

for-in above a dictionary

You can also iterate over a dictionary and access its key-value pairs. Each entry in the dictionary returns as an n-tuple (key, value) when iterates the dictionary, and you can decompose the parts of the n-tuple (key, value) as explicitly named constants for use in the body of the input loop:

```
let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
for (animalName, legCount) in numberOfLegs {
  print ("\ (animalName) has \ (legCount) legs")
}
// the cat has 4 legs
// the ant has 6 legs
// the spider has 8 legs
```

The contents of the dictionary are inherently disordered and its iteration does not guarantee the order in which they will be read. The order in which entries are entered in the dictionary does not define the order in which they are iterated.

2.1.6

What were the (animalName, legCount) variables in the previous example?

- n-tuple in which the key-value pair is stored when iterating the dictionary
- n-tuple in which the loop counter is stored when iterating the dictionary
- n-tuple in which random values are stored when iterating the dictionary

📖 2.1.7

for-in for closed ranges

You can use a for-in loop to iterate numeric ranges. This example goes through the range of 1 to 5:

```
for index in 1 ... 5 {
  print ("\(index) times 5 is \(index * 5)")
}
// 1 times 5 is 5
// 2 times 5 is 10
// 3 times 5 is 15
// 4 times 5 is 20
// 5 times 5 is 25
```

The loop is iterated in the range of numbers from 1 to 5 inclusive, as indicated by the use of the closed range operator (...). The index value is set to the first number in the range (1), and commands are executed inside the loop for all numbers in the range.

The index is a constant whose value is automatically set at the beginning of each iteration of the loop. Therefore, the index does not have to be declared before use. It is implicitly declared in a loop, without the need for the keyword declaration of *let*.

📖 2.1.8

Optional index in the loop

If you do not need the index value, you can ignore it by using an underscore instead of a variable name:

```
let base = 3
let power = 10
var answer = 1
for _ in 1 ... power {
  answer *= base
}
print ("\(base) ^ \(power) is \(answer)")
//Prints "3 ^ 10 is 59049
```

The above example calculates the value of one number to another (in this case 3 to 10). For this calculation, the individual values of the loop counter are unnecessary - the loop runs 10 times. The underscore (_) character used instead of a loop variable causes individual values to be ignored and does not provide access to the current value during each iteration of the loop.

2.1.9

Can we omit (leave out) the counter in the for loop?

- Yes
- No

2.1.10

for-in for open ranges

In some situations, you may not want to use closed ranges that include both endpoints. For example, drawing minutes marks on a dial. You want to draw 60 markers, starting with 0 minutes. Use the half-open range operator (.. <) to include the lower limit, but not the upper limit.

```
let minutes = 60
for tickMark in 0 .. <minutes {
  // render the check mark every minute (60 times)
}
```

2.1.11

Another for-in loop step

Some users only want to mark their fifth minute on their dial. Use the stride (from: to: by :) function to skip unwanted passes through the loop:

```
let minuteInterval = 5
for tickMark in stride (from: 0, to: minutes, by:
minuteInterval) {
  // draw check mark every 5 minutes (0, 5, 10, 15 ... 45, 50,
55)
}
```

For a closed range, use stride (from: through: by :):

```
let hours = 12
let hourInterval = 3
```

```
for tickMark in stride (from: 3, through: hours, by:
hourInterval) {
  // draw check mark every 3 hours (3, 6, 9, 12)
}
```

2.1.12

Which loop will run 5 times with values of 5, 7, 9, 11, 13?

- for a in stride(from: 5, through: 14, by: 2)
- for a in stride(from: 5, through: 12, by: 2)
- for a in stride(from: 5, through: 2, by: 13)

2.1.13

While loops

The while loop executes a set of statements until the condition becomes false. While loops are useful when the number of iterations before the first iteration begins. Swift provides two types of while loops:

- while - with a condition at the beginning of the loop
- repeat-while - with a condition at the end of the loop

While loop

The while loop begins by evaluating the condition. If the condition is true, the set of statements are repeated until the condition becomes false:

```
while condition {
  //body of the loop
}
```

2.1.14

Repeat-while loop

The second variation of the while loop (repeat-while) first passes through the loop block, then evaluates the loop condition. It then continues to repeat the loop until the condition is false:

```
repeat {
//body of the loop
} while condition
```

📝 2.1.15

The loop with the condition at the beginning is:

- while
- repeat-while

2.2 Statements

📖 2.2.1

Conditional code execution

It is often useful to execute different pieces of code based on certain conditions. You may want to run another piece of code when an error occurs, or you may see a message when the value is too high or too low. To do this, you condition parts of your code. Swift provides two ways to add conditional branches to your code:

if statement

switch command

You usually use the if statement to evaluate simple conditions with only a few possible results. The switch statement is more suitable for more complex conditions with multiple possible permutations, and is useful in situations where pattern matching can help select the appropriate branch of code to execute.

📖 2.2.2

A simple if statement

In its simplest form, the if statement has a single condition. Executes a set of commands only if this condition is true.

```
var temperature = -2
if temperature <= 0 {
  print ("It's very cold.")
}
// Prints "It's very cold."
```

The above example checks if the temperature is less than or equal to 0 degrees Celsius (freezing point of water). If so, a report will be printed. Otherwise, no message is printed and code execution continues after the closing brace of the if statement.

📖 2.2.3

If-else statement

The if statement can provide an alternative set of statements (else) for situations where the condition evaluates to false:

```
temperature = 12
if temperature <= 0 {
  print ("It's very cold.")
} else {
  print ("It's not that cold.")
}
// Prints "It's not that cold."
```

One of these two branches is always done. As the temperature rises to 12 degrees Celsius, it is no longer cold enough, the else branch is called.

📖 2.2.4

Complex conditions

Multiple conditions that share the same body can be written one after the other, separated by a comma. If any of the conditions are met, the result of the condition is considered true. If the list is long, you can write it on more lines:

```
let someCharacter: Character = "e"
switch someCharacter {
case "a", "e", "i", "o", "u":
  print ("\(someCharacter) is a vowel")
case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
   "N", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
  print ("\(someCharacter) is a consonant")
default:
  print ("\(someCharacter) is not a vowel or a consonant")
}
// Prints "e is a vowel"
```

📖 2.2.5

The if-elseif-else statement

You can concatenate multiple if statements to test other options:

```
temperature = 25
if temperature <= 0 {
  print ("It's very cold")
} else if temperature> = 25 {
  print ("It's really warm.")
} else {
  print ("It's not that cold.")
}
// Prints "It's really warm."
```

An additional condition that responds to high temperatures has been added here. The last else condition remains and prints an answer to any temperature that is neither too hot nor too cold.

However, the last condition else is optional and can be omitted:

```
temperature = 18
if temperature <= 0 {
  print ("It's very cold.")
} else if temperature> = 25 {
  print ("It's really warm.")
}
```

Because the temperature is not too low or too high to trigger the if or else if conditions, no message will be printed.

📖 2.2.6

The switch command

The switch command takes one value and compares it to several possible values. It then executes the appropriate block of code when the condition is met. The switch statement provides an alternative to the if statement for responding to multiple potential states.

In its simplest form, the switch statement compares a value with one or more values of the same type:

```
switch value {
```

```
case value1:
  respond to value1
case value2,
   value3:
  respond to value2 or value3
default:
  otherwise, do something else
}
```

Each switch statement consists of several possible cases, each of which begins with the case keyword. Each switch statement must be exhaustive. This means that each possible value of the considered type must correspond to one of the conditions. If it is not possible to specify all the options, you can define a default block to be executed for all values that are not listed. The default block must always be the last:

```
let someCharacter: Character = "z"
switch someCharacter {
case "a":
  print ("First letter of the alphabet")
case "z":
  print ("Last letter of the alphabet")
default:
  print ("Some other character")
}
// Prints "Last letter of the alphabet"
```

🕮 2.2.7

Switch differences from other languages

Unlike switch statements in C and Objective-C, switch statements in Swift do not have to contain a break statement at the end of each block.

Each block must contain at least one command. It is not valid to write the following code because the first block is empty:

```
let anotherCharacter: Character = "a"
switch anotherCharacter {
case "a": // Invalid, block has empty body
case "A":
  print ("Letter A")
default:
  print ("It's not a letter A")
}
```

```
// This returns a compilation error.
```

Unlike the switch command in C, this switch command does not match "a" and "A". Swift returns a compilation error that block "a": does not contain any executable statements. To perform one block for "a" and "A", combine these two values into a compound case and separate them with commas:

```
let anotherCharacter: Character = "a"
switch anotherCharacter {
case "a", "A":
  print ("Letter A")
default:
  print ("No letter A")
}
// Prints "Letter A"
```

🕮 2.2.8

Switch command and intervals

The switch command allows you to check a value using ranges:

```
let count = 62
switch count {
case 0:
  print (“nothing”)
case 1 .. <5:
  print ("pair")
case 5 .. <12:
  print ("several")
case 12 .. <100:
  print ("tens")
case 100 .. <1000:
  print ("hundreds")
default:
  print ("a lot")
}
// Prints "tens"
```

🕮 2.2.9

Switch command and n-tuples

You can use n-tuples to test multiple values in the same switch statement. Each n-tuple element can be tested against a different value or range of values. Alternatively, use an underscore (_) to match any possible value.

The following example takes a point (x, y), expressed as a simple n-tuple of type (Int, Int):

```
let somePoint = (1, 1)
switch somePoint {
case (0, 0):
  print ("\(somePoint) lies at the beginning")
case (_, 0):
  print ("\(somePoint) lies on the x-axis")
case (0, _):
  print ("\(somePoint) lies on the y-axis")
default:
  print ("\(somePoint) does not lie on any axis")
}
// Prints "(1, 1) does not lie on any axis"
```

Unlike C, Swift allows multiple switch conditions to take into account the same value or values. In fact, point (0, 0) could satisfy all conditions. However, if multiple matches are possible, the first match is always used. Point (0, 0) first corresponds to condition (0, 0), so all other matching conditions would be ignored.

2.2.10

Binding values in the switch statement

The switch statement can create temporary constants or variables for the values being compared. This behavior is called value binding because the values are bound to temporary constants or variables in the condition block:

```
let anotherPoint = (2, 0)
switch anotherPoint {
case (let x, 0):
  print ("on the x-axis with the value x: \(x)")
case (0, years):
  print ("on the y-axis with the value y: \(y)")
case let (x, y):
  print ("somewhere else in (\(x), \(y))")
}
// Prints "on the x-axis with x: 2
```

The three switch blocks declare placeholder constants x and y, which temporarily take one or both n-tuple values from another point. The first case (flight x, 0) corresponds to any point with a value of y 0 and assigns the value of point x to the temporary constant x. Similarly, the second case case (0, let y) corresponds to any point with a value of x 0 and assigns the y value of the point to the temporary constant y.

Once temporary constants are declared, they can be used within a block of condition code.

This switch command does not have a default block. The last case of case let (x, y) declares n-tuples of two placeholders, which can correspond to any value. Because anotherPoint is always an n-tuple of two values, this case corresponds to all possible remaining values, and when the switch statement is exhaustive, no default block is needed.

2.2.11

The switch-case-where command

The switch statement can use the where clause to check other conditions:

```
let yetAnotherPoint = (1, -1)
switch yetAnotherPoint {
case let (x, y) where x == y:
  print ("For point (\(x), \ y)) x == y is true")
case let (x, y) where x == -y:
  print ("For point (\(x), \(y)) x == -y is true")
case let (x, y):
  print ("(\(x), \(y)) is any point")
}
// Prints "For point (1, -1) x == -y is true"
```

The three switch blocks declare placeholder constants x and y, which temporarily take two n-tuple values from yetAnotherPoint. These constants are used as part of the where clause to create dynamic conditions. The block condition is true only if the condition of the where clause evaluates to true for this value.

2.2.12

Complex conditions with a value binding

Complex cases may also involve value bindings. All complex case patterns must include the same set of value constraints, and each constraint must obtain a value of the same type from all patterns in the case. This ensures that no matter which part of the complex case matches, the code in the body of the case can always have access to the value for the bindings, and that the value always has the same type.

```
let stillAnotherPoint = (9, 0)
switch stillAnotherPoint {
case (let distance, 0), (0, let distance):
  print ("On the axis, \(distance) from the beginning")
default:
  print ("Not on the axis")
}
// Print "On axis, 9 from the beginning"
```

2.2.13

What does the code print?

```
let point = (9, 1)
switch point {
case (let distance, 0), (0, let distance):
  print ("On the axis, \(distance) from the beginning")
default:
  print ("Not on the axis")
}
```

- Not on the axis
- On the axis, 9 from the beginning
- On the axis, 10 from the beginning

2.3 Other commands

2.3.1

Transition commands

These statements change the execution of code by transferring it from one block of code to another. Swift has five commands to control the transition:

- continue
- break

- fallthrough
- return
- throw

We will now describe the continue, break and fallthrough statements.

2.3.2

The continue statement

The continue statement tells the loop to interrupt what it is doing and start again at the beginning of the next iteration with the loop.

The following example removes all vowels and spaces from a lowercase string to create a mysterious phrase:

```
let puzzleInput = "great minds think alike"
var puzzleOutput = ""
let charactersToRemove: [Character] = ["a", "e", "i", "o",
"u", " "]
for character in puzzleInput {
  if charactersToRemove.contains (character) {
    continue
  }
  puzzleOutput.append (character)
}
print (puzzleOutput)
// Prints "grtmndsthnklk"
```

The above code calls the continue keyword whenever it matches a vowel or a space, which causes the current iteration of the loop to break immediately and jump directly to the beginning of the next iteration.

2.3.3

Which command interrupts the current iteration of the loop and starts the next one?

- continue
- break
- return

2.3.4

Break statement

The break statement immediately terminates the execution of the entire control flow statement. You can use the break statement inside a switch statement or inside a loop if you want to stop executing a switch statement or loop before otherwise.

2.3.5

Loop break

If the break statement is used inside a loop block, loop execution ends immediately and code execution continues after the loop (after the } character that ends the loop block). No further code from the current iteration of the loop is executed and no further iterations of the loop are executed.

2.3.6

Which command interrupts the currently running loop?

- break
- continue
- return

2.3.7

Interrupt the switch command

When the break statement is used inside a switch statement, execution of the switch block ends and code execution continues after the switch block (after the } character that ends the switch statement).

The break statement can be used to ignore one or more conditions in a switch statement. Because the switch in Swift must have all conditions defined and does not allow empty commands, it is sometimes necessary to intentionally terminate its execution with the break command:

```
let numberSymbol: Character = "3"
var possibleIntegerValue: Int?
```

```
switch numberSymbol {
case "1":
  possibleIntegerValue = 1
case "2":
  possibleIntegerValue = 2
case "3":
  possibleIntegerValue = 3
case "4":
  possibleIntegerValue = 4
default:
  break
}
if let integerValue = possibleIntegerValue {
  print ("The integer value \(numberSymbol) is
\(integerValue).")
} else {
  print ("An integer value for \(numberSymbol) could not be
found.")
}
// Prints “The integer value 3 is 3.
```

📖 2.3.8

Fallthrough statement

Unlike C, the switch statement in Swift only executes the first block of the condition, which is true (there is no need to break at the end of the condition block, as in C). If you need the C behavior of the switch statement, use the fallthrough statement to do so

```
let integerToDescribe = 5
var description = "Number \(integerToDescribe) is"
switch integerToDescribe {
cases 2, 3, 5, 7, 11, 13, 17, 19:
  description += "prime number and also"
  fallthrough
default:
  description += " an integer."
}
print(description)
// Prints "Number 5 is a prime number and also an integer."
```

The fallthrough keyword does not check other conditions specified in the switch. The code of the block of the following condition, or the block of the default, starts immediately, just like in the C language.

2.3.9

Which command can you use to change the behavior of a switch to a C-like behavior?

- fallthrough
- continue
- break

2.3.10

Marked (labeled) blocks

In Swift, you can nest loops and conditional statements in other loops and conditional statements to create complex flow control structures. However, loops and conditional statements can use break statements to terminate their execution prematurely. Therefore, it is sometimes useful to explicitly specify which loop or conditional statement you want to end with the break statement. Similarly, if you have multiple nested loops, it can be useful to explicitly specify which loop should be affected by the continue statement.

To achieve these goals, you can label a loop statement or a conditional statement.

To mark a command, add a naming label: to the same line as the command's introductory keyword. Here is an example of this while loop syntax, although the principle is the same for all loops and switch statements:

```
name: while condition {
  // loop block
  break name
}
```

2.3.11

How do we name a block of code with the word "name"?

- name: while ...
- while: name ...

- while { ... }: name

2.3.12

The Guard statement

The guard statement conditionally executes statements, similar to the if statement. The guard statement does not have a block in case the condition is true (the program is expected to continue on the next line). It must always contain an else block:

```
guard let name = person["name"] else {
  return
}
```

If this condition is not met, the code is executed inside the else branch. This branch must pass control to leave the block of code in which the guard command appears. It can do this with a control transfer statement, such as return, break, continue, or throw, or it can call an error function or method, such as fatalError(_: file: line:).

2.3.13

What does the code print:

```
let a = 1
guard let b = a else {
  print ("a has no value")
break
}
print (“\(a) = \(b)”)
```

- 1 = 1
- a has no value
- 1 = 2

2.3.14

Check system API availability

Swift has built-in support for checking the availability of the system API, which ensures that you do not accidentally call an API that is not available.

The compiler uses the availability information in the SDK to verify that all APIs used in your code are available in the deployment target specified by your project. Swift reports an error at compile time if you try to use an API that is not available.

📖 2.3.15

Conditional code block execution based on API availability

To conditionally execute a block of code, use the availability condition in the if or guard statement with #available. The condition will be evaluated while the program is running according to the OS version on which the application is currently running:

```
if #available (platform name version, ..., *) {
  commands to execute, if APIs are available
} else {
  backup commands to execute if APIs are not available
}
```

The availability condition contains a list of platform names and versions. You use platform names such as iOS, macOS, watchOS, and tvOS. In addition to specifying major version numbers, such as iOS 8 or macOS 10.10, you can enter minor version numbers, such as iOS 11.2.6 and macOS 10.13.3.

The following example shows how to run code on iOS 10 and later and macOS 10.12 and later. The last argument, *, is required and specifies that on any other platform, the if block will run on the minimum version specified in your project:

```
if #available (iOS 10, macOS 10.12, *) {
  // Use iOS 10 API on iOS and use macOS 10.12 API on macOS
} else {
  // Go back to the previous API for iOS and macOS
}
```

Types, Operators, Strings, Characters

Chapter 3

3.1 Constants and variables (without type)

📖 3.1.1

You have successfully mastered the motivational part of the lection. And you can start studying ..

But first you have to answer the question.

📝 3.1.2

Do you want to continue?

📖 3.1.3

Constants and variables associate a name with a value. You create a constant (for example, maximumNumberOfLoginAttempts or welcomeMessage) and associate it with a value of a specific type (for example, the number 10 or the string "Hello"). The value of the constant cannot be changed. Once set, it remains the same throughout the program. Hence a constant. The variable can be changed in the program and set to another value.

📖 3.1.4

Constants and variables must be defined before use. You define constants with the keyword **let** and variables with the keyword **var**. Here is an example of how constants and variables can be used to track the number of login attempts a user has made:

Example:

```
let maximumNumberOfLoginAttempts = 10
var currentLoginAttempt = 0
```

📝 3.1.5

Will the maximumNumberOfLoginAttempts from the paragraph before change during the program?

3.1.6

Does the constant change during the program?

3.1.7

Next, we can deal with an example:

```
let maximumNumberOfLoginAttempts = 10
var currentLoginAttempt = 0
```

Here, the maximum number of allowed login attempts is declared as a constant, because the maximum value never changes. That is, the let *maximumNumberOfLoginAttempts* = 10. The current logon attempt counter is declared as a variable because this value must be incremented after each failed login attempt. So var currentLoginAttempt = 0

You can declare multiple constants or multiple comma-separated variables in a single line:

```
var x = 0.0, y = 0.0, z = 0.0
```

If the stored value in your code does not change, always declare it as a constant with the keyword let. Use variables only to store values that need to be changed.

3.1.8

How do you define a value that never changes when the program runs

3.2 Constants and variables (with type)

3.2.1

When you declare a constant or variable, you can specify its type to make it clear what kind of values a constant or variable can contain. Type the type after the colon after the name of the constant or variable.

```
var welcomeMessage: String
```

Any text value can now be assigned to the welcomeMessage variable without error:

```
welcomeMessage = "Hello"
```

You can define multiple related variables of the same type on a single line, separated by commas, specifying one type after the name of the last variable:

```
var x, y: Double
```

In practice, variables and constants without a type are usually used. If you enter an initial value for a constant or variable, Swift can almost always derive the type to use for that constant or variable.

3.2.2

Constant and variable names can contain almost any character, including Unicode characters:

```
let π = 3.14159
let 你好 = "你好 世界"
let ???????? = "dogcow"
```

Constant and variable names cannot contain blanks, math symbols, arrows, private (or invalid) Unicode points, or line and frame drawing characters. They also cannot begin with a number, although numbers may be included elsewhere in the name.

Once you declare a constant or variable of a certain type, you cannot declare it again with the same name or change it to store values of another type. You also cannot change a constant to a variable or a variable to a constant.

You can change the value of an existing variable to another value of a compatible type. In this example, the friendlyWelcome value changes from "Hello!" To "Bonjour!":

```
var friendlyWelcome = "Hello!"
friendlyWelcome = "Bonjour!"
```

Unlike a variable, the value of a constant cannot be changed once it has been set. An attempt to change the value of a constant is reported as an error when compiling the code.

3.2.3

Fill in: var is used to define:

3.2.4

Fill in: let is used for definition

3.2.5

Is this listing correct?

```
let width = 100
width = 120
```

- yes
- no

3.3 Listing constants and variables

3.3.1

You can print the current value of a constant or variable using the print function

```
let friendlyWelcome = “Hello.”
print(friendlyWelcome)
// prints "Hello."
```

print is a global function that prints a value followed by the end of a line to the appropriate output. For example, in Xcode, print prints its output to the Xcode console. (The second function, print, performs the same task without appending the end of the line to the end of the value to be printed.)

The print function prints any string value that you pass to it:

```
print("This is a string")
// print "This is a string"
```

The print function can also print more complex log messages, containing text as well as values for constants and variables. Wrap the name of the constant or variable in parentheses and escape them with a backslash before the opening parenthesis

```
print("The current value of friendlyWelcome is
\(friendlyWelcome)")
// prints "The current value of friendlyWelcome is Hello."
```

📖 3.3.2

You can use comments to include non-executable text in your code as a note or reminder. When compiling code, the Swift compiler ignores comments.

The comments in Swift are very similar to the comments in C. One-line comments start with two slashes //

```
// this is a comment
```

Multiline comments begin with a slash followed by an asterisk / * and end with an asterisk followed by a slash * /

```
/ * this is also a comment,
but written on more lines * /
```

Unlike multiline comments in C, multiline comments in Swift can be nested in other multiline comments. You write nested comments by running a block of multiline comments and then running the second multiline comment within the first block. The second block is then closed, followed by the first block:

```
/ * this is the beginning of the first multiline comment
/ * this is the second nested multiline comment * /
this is the end of the first multiline comment * /
```

Nested multiline comments make it quick and easy to comment on large blocks of code, even if the code already contains multiline comments.

📖 3.3.3

Unlike many other languages, Swift does not require you to write a semicolon after each statement in your code, although you can do so if you wish. However, semicolons are required if you want to write multiple separate statements on a single line

```
let cat = "????"; print (cat) // print "????"
```

Q:

Which entry is correct:

```
let width = 100 let height = 80; - Incorrect
let width = 100; let height = 80 - Correct
```

3.3.4

Print the value of the variable *quantity* in the form "number: xx" to the console

3.3.5

Is the notation /* first /* in the notation */ note */ correct?

3.3.6

Which entry is correct:

- let width = 100 let height = 80;
- let width = 100; let height = 80

3.4 Types

3.4.1

Integer

Integers are numbers without a decimal component, such as 53 and -16. Integers can be signed (positive, zero or negative) or unsigned (positive or zero). Swift provides signed and unsigned integers in 8, 16, 32 and 64 bit forms. These integers follow a C-like naming convention: an 8-bit unsigned integer is of type UInt8 and a 32-bit unsigned integer is of type Int32. Like all types in Swift, these integer types have uppercase and lowercase letters.

3.4.2

Which entry is correct?

- let a: Int = 10
- var b: Uint = 20

3.4.3

Ranges of integers

You can access the minimum and maximum value of each integer type with its min and max properties

```
years minValue = UInt8.min // minValue is equal to 0 and is of
type UInt8
years maxValue = UInt8.max // maxValue is equal to 255 and is
of type UInt8
```

The values of these properties have the same numeric type (for example, UInt8 in the example above) and can therefore be used in expressions along with other values of the same type.

```
Int
```

In most cases, you don't have to select a specific integer size in your code. Swift provides another integer type of Int, which is the same size as the native word size of the current platform:

On a 32-bit platform, Int is the same size as Int32.

On a 64-bit platform, Int is the same size as Int64.

If you do not need to work with a specific integer size, always use Int for integer values in the code. This helps ensure code consistency.

```
UInt
```

Swift also provides an unsigned integer type, UInt, which is the same size as the native word size of the current platform:

On a 32-bit platform, UInt is the same size as UInt32.

On a 64-bit platform, UInt is the same size as UInt64.

It is recommended to use Int as much as possible, although it is clear in advance that the values will only be positive.

3.4.4

Is it recommended to use UInt for positive number values in Swift?

📖 3.4.5

Floating point numbers

Floating point numbers are numbers with a decimal component, such as 3.141592, 0.5, and -329.26.

Floating point types can contain values with a much larger range than integer types. Swift provides two signed types of floating point numbers:

Double represents a 64-bit floating point number.

The float value represents a 32-bit floating point number.

Double has an accuracy of at least 15 decimal places, while Float accuracy can be only 6 decimal places. The appropriate floating point type to use depends on the nature and range of values you must work with in your code. In situations where both types can be used, the Double type is preferred.

📝 3.4.6

Which type of floating point is preferred in Swift?

📖 3.4.7

Type safety and type derivation

Swift is a type-safe language. If part of your code expects a string, you can't pass it to Int by mistake.

Because Swift is type-safe, it performs type checks when compiling code and marks all mismatched types as errors. This will allow you to catch and fix bugs as soon as possible in the development process.

Type checking helps you avoid errors when working with different types of values. However, this does not mean that you must specify the type of each constant and variable that you declare. If you do not specify the desired value type, Swift uses a type derivation to determine the appropriate type. Type Derivation allows the compiler to automatically derive the type of a particular expression when it compiles your code, simply by examining the values you enter.

Due to type derivation, Swift requires far fewer type declarations than languages such as C or Objective-C. Constants and variables are still explicitly specified, but most of the work with specifying their type is done for you.

Type derivation is especially useful when you declare a constant or variable with an initial value. This is often done by assigning a literal value to a constant or variable where you declare it.

For example, if you assign a literal value of 42 to a new constant without telling you what type it is, Swift concludes that you want the constant to be Int because you initialized it with a number that looks like an integer:

```
let myValue = 42
// myValue is derived to be of type Int
```

Similarly, if you do not specify a type for a floating-point literal, Swift infers that you want to create a Double:

```
let pi = 3.141592
// pi is derived to be of type Double
```

When deriving a floating-point number type, Swift always chooses Double instead of Float.

If you combine integer and floating-point literals in an expression, the Double type is derived from the context.

```
let calculatedPi = 3 + 0.141592
// calculatedPi is also derived to be of type Double
```

The value 3 has no explicit type per se, so the appropriate Double output type is derived from the presence of a floating-point literal as part of the addition.

3.4.8

What type will the variable be defined as var num = 1.65

3.4.9

Numeric literals

Integer literals can be written as:

- Decimal number without prefix
- Binary number with prefix 0b
- An octal number prefixed with 0o
- Hexadecimal number prefixed with 0x

All these integer literals have a decimal value of 17

```
let decimalInteger = 17
let binaryInteger = 0b10001 // 17 in binary notation
let octalInteger = 0o21 // 17 in octal notation
let hexadecimalInteger = 0x11 // 17 in hexadecimal notation
```

Floating point literals can be decimal (without a prefix) or hexadecimal (with a prefix 0x). It must always have a number (or hexadecimal number) on both sides of the decimal point. They may also have an optional exponent denoted by uppercase or lowercase letters *e* for decimal numbers or uppercase or lowercase letters *p* for hexadecimal numbers.

For decimal numbers with the exp exponent, the base number is multiplied by 10^exp

1.25e2 means 1.25 × 10^2 = 125.0.

1.25e-2 means 1.25 × 10^-2 = 0.0125.

For hexadecimal numbers with the exp component exp, the base number is multiplied by 2^exp

0xFp2 means 15 × 2^2 = 60.0.

0xFp-2 means 15 × 2^-2 = 3.75.

All of these floating-point literals have a decimal value of 12.1875

```
let decimalDouble = 12.1875
let exponentDouble = 1.21875e1
let hexadecimalDouble = 0xC.3p0
```

Numeric literals may contain additional formatting to make them easier to read. Both integers and floating numbers can be filled with extra zeros and can contain underscores to help with readability. Neither type of formatting affects the base value of a literal

```
let paddedDouble = 000123,456
let oneMillion = 1_000_000
let justOverOneMillion = 1_000_000.000_000_1
```

3.4.10

They can contain values of constants and variables with the _ character

3.5 Casting

📖 3.5.1

Numeric type conversion

Use the Int type for all integer constants and variables in the code, even if they are known to be nonnegative. Using the default integer type in everyday situations means that integer constants and variables will be compatible with each other. Use other types of integers only if they are specifically needed for the task, because data is available from an external source of a given size, memory usage, or other necessary optimization.

📝 3.5.2

Is it recommended to use the Int type for all integer constants and variables in the code?

📖 3.5.3

Integer conversion

The range of numbers that can be stored in an integer constant or variable is different for each numeric type. The constant or variable Int8 can store numbers between -128 and 127, while the constant or variable UInt8 can store numbers between 0 and 255. A number that does not fit in a constant or variable of type integer is reported as a translation error.

```
let cannotBeNegative: UInt8 = -1
// UInt8 cannot store negative numbers, so an error will be
reported
let tooBig: Int8 = Int8.max + 1
// Int8 cannot store a number greater than its maximum value
```

Because each numeric type can store a different range of values, you must convert numbers on a case-by-case basis.

To convert one particular number type to another, you initialize a new number of the desired type with an existing value. In the example below, the two thousand constant is of type UInt16, while the constant of one is of type UInt8. They cannot be added directly because they are not of the same type. Instead, this example calls UInt16 (one) to create a new UInt16 initialized with a value of one and uses that value instead of the original.

```
let twoThousand: UInt16 = 2_000
let one: UInt8 = 1
years twoThousandAndOne = twoThousand + UInt16 (one)
```

Because both sides of the addition are now of type UInt16, the sum is allowed. The output constant (twoThousandAndOne) is derived as type UInt16 because it is the sum of two UInt16 values.

3.5.4

Why convert constants or variables consistently before adding them up?

- Because they may have a different number range
- Because the right result would not be displayed
- Because they may have different data types

3.5.5

Integer and real number conversion

Conversions between integers and real numbers must be made explicitly

```
let three = 3
let pointOneFourOneFiveNineTwo = 0.141592
let pi = Double (three) + pointOneFourOneFiveNineTwo
// pi equals 3.14159 and is derived to be of type Double
```

Here, the value of constant three is used to create a new value of type Double, so that both sides of the addition are of the same type. Without this on-site conversion, the addition would not be allowed.

Floating point conversion to an integer must also be done explicitly. The integer can be initialized with Double or Float

```
let integerPi = Int(pi)
// integerPi equals 3 and is derived to be of type Int
```

If real numbers are used to initialize a new integer value, the decimal places are removed. This means that 4.75 becomes 4 and -3.9 becomes -3.

📖 3.5.6

Type aliases

Type aliases define an alternate name for an existing type. You define type aliases with the typealias keyword. Type aliases are useful when you want to reference an existing type with a name that is more contextually appropriate, such as when working with data of a specific size from an external source.

```
typealias AudioSample = UInt16
```

Once you define an alias type, you can use the alias anywhere you can use the original name

```
var maxAmplitudeFound = AudioSample.min
// maxAmplitudeFound is now 0
```

Here, AudioSample is defined as an alias for UInt16. Because this is an alias, calling AudioSample.min actually calls UInt16.min, which provides an initial value of 0 for the maxAmplitudeFound variable.

📝 3.5.7

What logical constants do we use?

- true, false
- false = false
- true = true

📖 3.5.8

Logical type

Swift has a basic logical type called Bool. Logical values are called logical because they can always be true or false. Swift provides two logical constant values, true and false

```
let orangesAreOrange = true
let turnipsAreDelicious = false
```

The orangesAreOrange and turnipsAreDelicious types were derived as Bool from the fact that they were initialized using Boolean literal values. As with Int and Double before, you don't have to declare constants or variables as Bool if you set them to true or false once you've created them. Type derivation helps make Swift code shorter and more readable.

3.6 N-tuples, types, bonds

🕮 3.6.1

N-tuple

N-tuples group multiple values into one compound value. The values in the n-tuple can be of any type and do not have to be of the same type as the others.

In this example (404, “Not Found”), it is an n-tuple that describes the HTTP status code. The HTTP status code is a special value returned by the webserver whenever you request a web page. If you request a non-existent webpage, a 404 Not Found status code is returned.

```
let http404Error = (404, "Not Found")
// http404Error is of type (Int, String) and equals (404, "Not
Found")
```

Said n-tuple groups Int and String. It contains a number and a "readable" description. It can be described as n-tuples of type (Int, String).

You can create n-tuples from any type of permutation and they can contain as many different types as you want. There is nothing stopping you from having an n-type (Int, Int, Int) or (String, Bool) or any other permutation you require.

You can break down the contents of an n-tuple into separate constants or variables, which you then access as usual:

```
let (statusCode, statusMessage) = http404Error
print("Status code is \(statusCode)")
// prints "Status code is 404"
print("Status message is \(statusMessage)")
// prints "Status message not found
```

If you only need some of the values of the n-tuple, ignore the parts of the n-tuple with the underscore _ when you decompose the n-tuple:

```
let (justTheStatusCode, _) = http404Error
print("Status code is \(justTheStatusCode)")
// prints "Status code is 404"
```

Alternatively, you can access the individual values of the elements in the n-tuple using index numbers starting from zero:

```
print("Status code is \(http404Error.0)")
// prints "Status code is 404"
print("Status message is \(http404Error.1)")
// prints "Status message not found"
```

When an n-tuple is defined, you can name individual elements in the n-tuple:

```
let http200Status = (statusCode: 200, description: “OK”)
```

If you name elements in an n-tuple, you can use element names to access the values of those elements:

```
print("Status code is \(http200Status.statusCode)")
// prints "Status code is 200"
print("Status message is \(http200Status.description)")
// prints "Status message is OK"
```

N-tuples are particularly useful as return values of functions. A function that attempts to load a Web page may return an n-tuple (Int, String) type, which describes the success or failure of the page loading. By returning an n-tuple with two different values, each of which has a different type, the function provides more useful information about its result than if it could return only one value of a single type.

N-tuples are useful for temporary groups of related values. They are not suitable for creating complex data structures.

3.6.2

Must the N-tuples be of the same type as the others?

- no
- yes

3.6.3

Optional types

You use optional types in situations where a value may be missing. Optional says there is a value equal to x or no value at all.

The concept of optional items in C or Objective-C does not exist. The closest thing to Objective-C is the ability to return nil from a method that would otherwise return an object, where nil means "absence of a valid object." However, this only works for objects - it doesn't work for structures, basic C types, or enumeration values. For these types, Objective-C methods typically return a special value (such as NSNotFound) that indicates the absence of a value. This approach assumes that the calling method knows that there is a special value to test and remembers to check it. Optional Swift types allow you to indicate the absence of a value for any type without the need for special constants.

Here is an example of how to use options to deal with the absence of value. A String type has a method called toInt that attempts to convert a String value to an Int value. However, not every string can be converted to an integer. The string "123" can be converted to a numeric value of 123, but the string "hello" does not have an obvious numeric value to convert to.

The following example uses the toInt () method to try to convert a string to Int:

```
let possibleNumber = "123"
let convertedNumber = possibleNumber.toInt ()
// the converted number is derived from the type "Int?" or
"optional Int"
```

Because the toInt() method may fail, it returns an optional Int, not Int. The optional Int is written as Int ?, not Int. The question mark indicates that the value it contains is optional, which means that it can contain some Int value or no value at all. (It can't contain anything else, such as a Bool value or a String value. It's either Int or nothing at all.)

3.6.4

Which type is optional?

- Int
- Int?
- Int = nil

3.6.5

nil

To set an optional variable to no value, assign it a special value of nil

```
var serverResponseCode: Int? = 404
```

```
// serverResponseCode contains the actual Int value of 404
serverResponseCode = nil
// serverResponseCode currently has no value
```

nil cannot be used with non-optional constants and variables. If a constant or variable in your code must work with the absence of a value, always declare it as an optional value of the appropriate type.

If you define an optional variable without entering a default value, the variable is automatically set to nil for you

```
var surveyAnswer: String?
// surveyAnswer is automatically set to nil
```

Swift nil is not the same as nil in Objective-C. In Objective-C, nil is a pointer to a non-existent object. There is no nil indicator in Swift - it is the absence of a value of a certain type. You can set any type, not just object types.

3.6.6

How do you set an optional variable to no value (nil)?

- by assigning a special value
- by converting to nill
- setting to nil

3.6.7

Forced value expansion

You can use the if statement to determine if an optional variable contains a value by comparing the optional value with nil. You perform this comparison with the "equals" operator (==) or with the "does not equal" (! =) operator.

If an optional variable has a value, it is considered "not equal" to nil

```
if convertedNumber != nil {
  print ("The converted number contains some integer value.")
}
// prints "the converted number contains some integer value.
```

Once you are sure that an optional variable contains a value, you can access its value by adding an exclamation point! at the end of the optional variable name. The exclamation point actually says, “I know that this optional variable definitely has its value; please use it."

```
if convertedNumber != nil {
  print("The converted number has an integer value of \(convertedNumber!).")
}
// prints "the converted number has an integer value of 123."
```

If you use ! to access a non-existent optional value a runtime error is triggered. Always make sure that the optional value contains a non-nil value before use.

3.6.8

If the optional variable has a value

- is considered "not equal" or "not equal!"
- is considered "not equal" nil
- is considered "equal" nil

3.6.9

Optional binding

Use the optional constraint to determine if the optional variable contains a value, and if so, to make it available as a temporary constant or variable. An optional constraint can be used with if and while statements to check a value inside an optional element and to extract that value into a local constant or variable.

Type an optional binding for the if statement as follows:

```
if let constantName = someOptional {
  ...
}
```

You can override the possibleNumber example from the Optionals section and use an optional constraint rather than enforcing a value

```
if let actualNumber = possibleNumber.toInt () {
  print("\'\(possibleNumber)\' has an integer value \(actualNumber)")
} else {
  print("\'\(possibleNumber)\' cannot be converted to an
integer")
}
// prints "'123' has an integer value 123"
```

This code can be read as:

"If the optional Int returned by possibleNumber.toInt contains a value, set a new constant called actualNumber to the value contained in the optional."

If the conversion is successful, the actualNumber constant will be available for use in the first branch of the if statement. It has already been initialized with the value contained in the optional value, so there is no need to use ! for access to its value. In this example, the current number is used to print the conversion result.

You can use constants and variables with optional binding. If you wanted to manipulate the value of currentNumber in the first branch of the if statement, you could type, *if var actualNumber* instead, and the value contained in the optional value would be made available as a variable.

Multiple optional bindings can appear in a single if statement as a comma-separated list of assignment expressions.

```
if let constantName = someOptional, anotherConstantName =
someOtherOptional {
  ...
}
```

3.6.10

Use an optional constraint to determine if the optional variable contains a value. Which commands can be used with?

- “and” & “or”
- “nil” & “if” or “nil” & “while”
- ”if” & “while”

3.7 Errors, statements, preconditions

3.7.1

Default unpacked values

Sometimes it is clear from the program structure that an optional variable will always have a value when this value is set for the first time. In these cases, it is useful to eliminate the need to check and expand the value of an optional variable on each access, because it can be safely assumed that it has a value at all times.

These types of optional variables are defined as implicitly unpackaged optional variables. To write an expanded option, place an exclamation point (String!) instead of a question mark (String?) after the type you want to mark as optional.

The default expanded option is a normal background option, but can also be used as an optional value without having to expand the option each time you access it. The following example shows the difference in behavior between an optional string and the implicitly "unwrapped optional string when accessing their wrapped value as an explicit string:

```
let possibleString: String? = "Optional string."
let forcedString: String = possibleString! // requires an
exclamation mark
let assumedString: String! = "Optional string expanded by
default."
let implicitString: String = assumedString // no exclamation
point required
```

You can think of an implicitly expanded optional variable as providing permission to automatically expand an optional variable each time it is used. Instead of inserting an exclamation point after a variable name each time you use it, place an exclamation point after the variable type when you declare it.

If you try to access an implicitly expanded optional variable if it does not contain a value, a runtime error is raised. The result is exactly the same as placing an exclamation point after a normal optional variable that does not contain a value.

You can still treat an implicitly expanded optional variable as a normal optional variable to verify that it contains the value:

```
if assumedString != nil {
  print(assumedString)
}
// prints "Optionally expanded optional string."
```

You can also use an implicitly expanded optional variable with an optional binding to check and extract its value in a single statement:

```
if let definiteString = assumedString {
  print(definiteString)
}
// prints "Optional string extended by default."
```

Do not use an implicitly expanded optional variable if there is a possibility that the variable may contain nil at a later time. If you need to check the nil during the life cycle of the variable, always use the normal optional type.

📖 3.7.2

Error handling

By processing errors, you respond to error conditions that your program may encounter during execution.

```
func canThrowAnError() throws {
  // this function may or may not cause an error
}
do {
  try canThrowAnError()
  // no error was raised, the run continues with another line
} catch errorCode1 {
  // error 1 occurred
catch errorCode2 {
  // error 2 occurred
}
```

📖 3.7.3

Assertions

Options allow you to check for values that may or may not exist and to write code that elegantly copes with the absence of a value. However, in some cases, it's simply not possible for your code to continue executing if the value doesn't exist or if the value you enter doesn't meet certain conditions. In these situations, you can activate a statement in your code that terminates code execution and provides an opportunity to debug the cause of a missing or invalid value.

Debugging by assertion

If the condition evaluates to true, code execution continues as usual; if the condition evaluates to false, code execution ends and your application terminates.

If your code runs a statement at runtime in a debugging environment, such as Xcode, you can see exactly where the invalid state occurred. The statement will also allow you to provide a suitable debugging report.

Write the statement by calling the global function assert(_: _: file: line:). You pass an expression to the assert function that evaluates to true or false, and a message that should appear if the result of the condition is false:

```
let age = -3
assert(age> = 0, "A person's age cannot be less than zero")
// this causes the statement to run because age is not >= 0
```

In this example, code execution will continue only if age> = 0 evaluates to true, that is, if the age value is non-negative. If the age value is negative, as in the code above, then age> = 0 evaluates to false and the statement is activated and the application is terminated.

The assertion message can be omitted if necessary, as in the following example:

```
assert (age> = 0)
```

Use the statement whenever the condition has the potential to be false, but it must definitely apply in order for your code to continue executing.

If the check has already taken place, use the assertionFailure (_: file: line:) function to indicate that the statement failed:

```
if age > 10 {
  print("You can ride the roller-coaster or the ferris
wheel.")
} else if age >= 0 {
  print("You can ride the ferris wheel.")
} else {
  assertionFailure("A person's age can't be less than zero.")
}
```

3.7.4

If the condition does not evaluate to true,

- code execution continues as usual
- code execution ends and your application terminates
- the application prints the global function false

3.7.5

Preconditions

Use preconditions whenever a condition has the potential to be false, but for your code to continue executing, it must be true. For example, use a precondition to check that the subscript is not out of bounds or to check that a valid value has been passed to the function.

To write the precondition, call the precondition(_: _: file: line:) function. You pass an expression to this function that evaluates to true or false, and a message that appears if the result of the condition is false. For example:

```
// When implementing subscript ...
precondition(index> 0, "Index must be greater than zero.")
```

You can also call the preconditionFailure (_: file: line :) function to indicate that an error has occurred.

3.7.6

Which situation can occur when using the precondition so that the code can continue

- The precondition must apply
- On the contrary, the precondition doesn't need to apply
- It depends on the definition of the precondition whether or not it must apply

3.8 Operators 1

3.8.1

Terminology

Operators are unary, binary, or ternary: Unary operators operate on a single target (for example, -a). Unary operators can appear before their target (for example !b) or behind their target (for example i++). Binary operators work on two targets (for example, 2 + 3) and are located between their two targets. Ternary operators operate on three targets. Like C, Swift has only one ternary operator, the ternary conditional operator (a?b:c). The values that affect operators are operands. In the expression 1 + 2, the symbol + is a binary operator and its two operands are the values 1 and 2.

3.8.2

Unary operators can appear:

- before or after their target
- before their target
- after their target

3.8.3

Assignment operator

The assignment operator (a = b) initializes or updates the value of a with the value of b:

```
let b = 10
var a = 5
a = b
// and is now equal to 10
```

If there is an assignment of a n-tuple with multiple values on the right side, its elements can be decomposed into several constants or variables at once:

```
let (x, y) = (1, 2)
// x is equal to 1 and y is equal to 2
```

3.8.4

Choose the correct answer: Assignment operator

a) does not initialize the value

b) does not update the value

c) initializes or updates the value

- does not initialize the value
- does not update the value
- initializes or updates the value

3.8.5

The assignment operator does not return a value

Unlike the assignment operator in C and Objective-C, the assignment operator in Swift itself does not return a value. The following statement is not valid:

```
if x = y {
 // this is not valid because x = y does not return a value
}
```

This function prevents the assignment operator (=) from being used accidentally instead of the comparison operator (==).

3.8.6

Arithmetic operators

Swift supports four standard arithmetic operators for all types of numbers:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)

```
1 + 2 // equals 3
5 - 3 // equals 2
2 * 3 // equals 6
10.0 / 2.5 // equals 4.0
```

Unlike C arithmetic operators and Objective-C Swift, arithmetic operators do not allow values to overflow by default.

The addition operator can also be used to concatenate strings:

```
"Hello," + "world" // equals "Hello, world"
```

3.8.7

Which characters can be operators?

a) @, $

b) +, -

c) +, -, =

- @, $
- +, -
- +,-,=

📖 3.8.8

The remainder operator after division

The remainder operator (a % b) finds out how many multiples of b fit in a, and returns the remaining value (remainder).

```
9 % 4 // equals 1
-9 % 4 // equals -1
```

Unlike the remainder operator in C and Objective-C, the Swift remainder operator can also work with floating-point numbers:

```
8% 2.5 // equals 0.5
```

In this example, 8 divided by 2.5 equals 3, with a remainder of 0.5, so the remainder operator returns a Double value of 0.5.

📝 3.8.9

Which notation for the *remainder* operator returns 3?

- 5%5
- 15%5
- 28%5

3.9 Operators 2

📖 3.9.1

Unary operator minus

The sign of a numeric value can be changed using a prefix - known as the unary minus operator:

```
let three = 3
let minusThree = -tree
// minusThree equals -3
let plusThree = -minusThree
// plusThree equals 3 or "minus minus three"
```

The unary operator minus (-) is appended directly before the value, without spaces. The unary operator plus (+) returns the value without change:

```
let minusSix = -6
let alsoMinusSix = +minusSix
// alsoMinusSix equals -6
```

3.9.2

The unary operator minus (-) is appended before the value:

- with space
- without space
- with or without space

3.9.3

Merged assignment operators

Like C, Swift provides compound assignment operators that combine an assignment (=) with another operation. One example is the add assignment operator (+=):

```
var a = 1
a += 2
// and is now equal to 3
```

The expression a += 2 is an abbreviation for a = a + 2. Addition and assignment are effectively combined into one operator, which performs both tasks simultaneously. Compound assignment operators do not return a value. For example, you cannot write years b = a += 2. This behavior is different from the increase and decrease operators listed above.

3.9.4

Comparison operators

Swift supports all standard C comparison operators:

- equals (a == b)
- unequal (a != b)
- greater than (a > b)
- less than (a < b)
- greater than or equal to (a >= b)
- less than or equal to (a <= b)

Swift also provides two identity operators (=== and !==) that you use to test whether two object references point to the same object instance.

Each of the comparison operators returns a Bool value that indicates whether the statement is true or not:

```
1 == 1 // true because 1 equals 1
2 != 1 // true, because 2 is not equal to 1
2 > 1 // true because 2 is greater than 1
1 < 2 // true because 1 is less than 2
1 >= 1 // true because 1 is greater than or equal to 1
2 <= 1 // false because 2 is not less than or equal to 1
```

Comparison operators are often used in conditional statements, such as the if statement:

```
let name = "world"
if name == "world" {
  print ("hello world")
} else {
  print ("sorry \(name), but I don't know you")
}
// prints "hello, world" because the name really equals
"world"
```

3.9.5

Is the condition for the equality of two variables written correctly?

```
if a = b
```

- no
- yes

📖 3.9.6

Ternary conditional operator

The ternary conditional operator is a special operator with three parts, which has the form question?answer1:answer2. It is an abbreviation for evaluating one of two terms based on whether the question is true or false. If the question is true, answer1 is executed and returns its value; otherwise, it executes response2 and returns its value.

The ternary conditional operator is an abbreviation for the code below:

```
if question {
  answer1
} else {
  answer2
}
```

Here is an example that calculates the height of a table row. The line height should be 50 points higher than the content height if the line has a header, and 20 points higher if the line does not have a header:

```
let contentHeight = 40
let hasHeader = true
let rowHeight = contentHeight + (hasHeader?50:20)
// rowHeight equals 90
```

rowHeight can be briefly set to the correct value on a single line of code.

Use the ternary conditional operator carefully. Its brevity can lead to hard-to-read code. Do not combine multiple instances of a ternary conditional operator into a single compound statement.

📝 3.9.7

What will be the value of the constant b?

```
let a = 10
let b = a <10 ? "smaller": "bigger"
```

- smaller
- bigger

📖 3.9.8

Operator ??

The operator (a ?? b) returns the optional value of a, if a contains the value, or returns the default value of b. The expression a is always of the optional type. The expression b must correspond to type a.

The zero fusion operator is an abbreviation for the code below:

The following example uses the zero fusion operator to choose between a default color name and an optional user-defined color name:

```
let defaultColorName = "red"
var userDefinedColorName: String? // default value nil
var colorNameToUse = userDefinedColorName ?? defaultColorName
// userDefinedColorName is nil, so colorNameToUse is set to
the default value of "red"
userDefinedColorName = "green"
colorNameToUse = userDefinedColorName ?? defaultColorName
// userDefinedColorName is not nil, so colorNameToUse is set
to "green"
```

📖 3.9.9

Range operators

Swift includes two range operators to express a range of values.

Closed range operator

The closed range operator (a ... b) defines the range from a to b and includes the values a and b. The value of a cannot be greater than b. The closed range operator is useful in an iteration in which you want to use all values:

```
for index in 1 ... 5 {
  print ("\(index) times 5 is \(index * 5)")
}
// 1 times 5 is 5
// 2 times 5 is 10
// 3 times 5 is 15
// 4 times 5 is 20
// 5 times 5 is 25
```

3.9.10

How is the range written?

- 1.5
- 1..5
- 1...5

3.9.11

Semi-open range operator

The semi-open range operator (a .. <b) defines the range from a to b, but does not include b. It is said to be semi-open because it contains its first value, but not its final value. As with the closed range operator, the value of a must not be greater than b. If the value of a is equal to b, the resulting range will be empty.

Semi-open ranges are especially useful when working with zero-based lists, such as fields, where it's useful to count up to (but without) the length of the list:

```
let names = ["Anna", "Alex", "Brian", "Jack"]
let count = names.count
for i in 0..<count {
  print ("Person \(i + 1) is named \(names [i])")
}
// Person 1 is named Anna
// Person 2's name is Alex
// Person 3's name is Brian
// Person 4's name is Jack
```

3.10 Operators 3

3.10.1

One-sided ranges

The closed range operator has an alternative form for ranges that continue in one direction if possible - for example, a range that includes all elements of the array from index 2 to the end of the array. In these cases, you can omit the value on one side of the range operator. This type of range is called a one-sided range because the operator has a value on one side only.

```
for name in names [2...] {
  print(name)
}
// Brian
// Jack
for name in names [...2] {
  print(name)
}
// Anna
// Alex
// Brian
```

You can also write:

```
for name in names [..<2] {
  print(name)
}
// Anna
// Alex
You can also check to see if a single-sided range has a
specific value:
let range = ... 5
range.contains (7) // false
range.contains (4) // true
range.contains (-1) // true
```

📖 3.10.2

Logical operators

Logical operators modify or combine the logical values true and false. Swift supports three standard C-derived logical operators:

- logical NO (!a)
- logical A (a && b)
- logical OR (a || b)

📖 3.10.3

Logical operator NOT

The logical operator NOT (!A) inverts the logical value so that true becomes false and false becomes true. The logical operator NO is the prefix operator and appears immediately before the value it is working on, without spaces:

```
let allowEntry = false
if !allowEntry {
  print("ACCESS DENIED")
}
// prints "ACCESS DENIED"
```

3.10.4

Logical operator AND

The logical operator AND (a && b) creates logical expressions, where both values must be true for the entire expression to be true. If any value is false, the entire expression will also be false. If the first value is false, the second value will not be evaluated because the whole expression cannot be true:

```
let enteredDoorCode = true
let passedRetinaScan = false
if enteredDoorCode && passedRetinaScan {
  print("Welcome!")
} else {
  println("ACCESS DENIED")
}
// prints "ACCESS DENIED"
```

3.10.5

Command:

```
let boolVal1 = true
let boolVal2 = false
if boolVal1 && boolVal2 {
  print(“Condition met”)
}
else {
  print (“Condition not met”)
}
```

prints:

- Condition met
- Condition not met

📖 3.10.6

Logical operator OR

The logical operator OR (a || b) creates logical expressions in which only one of the two values must be true for the entire expression to be true.

If the left side of the logical OR expression is true, the right side is not evaluated because it cannot change the result of the overall expression:

```
let hasDoorKey = false
let knowOverridePassword = true
if hasDoorKey || KnowOverridePassword {
  print("Welcome!")
} else {
  print("ACCESS DENIED")
}
// prints "Welcome!"
```

📝 3.10.7

Command:

```
let boolVal1 = true
let boolVal2 = false
if boolVal1 || boolVal2 {
  print(“Condition met”)
}
else {
  print(“Condition not met”)
}
```

prints:

- Condition met
- Condition not met

📖 3.10.8

Combination of logical operators

You can combine multiple logical operators to create longer compound expressions:

```
if enteredDoorCode && passedRetinaScan || hasDoorKey ||
knowOverridePassword {
  print("Welcome!")
} else {
  println("ACCESS DENIED")
}
// prints "Welcome!"
```

This example uses multiple && and || operators to create a longer compound expression. && and || operators only work on two values, so they are actually three smaller expressions concatenated together.

Logical operators in Swift - && and || are evaluated from left to right.

3.10.9

Explicit parentheses

Sometimes it is useful to use parentheses, although this is not necessary to make the source code easier to read. In the door access example before, it is useful to add parentheses around the first part of a compound expression to make its intent explicit:

```
if (enteredDoorCode && passedRetinaScan) || hasDoorKey ||
knowOverridePassword {
  print("Welcome!")
} else {
  print("ACCESS DENIED")
}
// prints "Welcome!"
```

It is clearly stated in parentheses that the first two values are considered in the overall logic to be part of a separate possible state. The output of the compound expression does not change, but the overall intent is clearer. Readability is always preferred to brevity: use parentheses where they help you clarify your intentions.

3.10.10

Are the following conditions equivalent?

```
if enteredDoorCode && passedRetinaScan || hasDoorKey ||
knowOverridePassword
if (enteredDoorCode && passedRetinaScan) || hasDoorKey ||
knowOverridePassword
```

- yes
- no

3.11 Strings and characters

🕮 3.11.1

Strings and characters

A string is an ordered array of characters, such as "hello, world," or "swift." Swift strings are represented by the String type, which in turn represents an array of Character values.

The String and Character types provide a quick way to work with unicode text. The syntax for creating and manipulating strings is simple and readable. The text is concatenated simply by the + operator and the same rules of constants (let) and variables (var) apply to them.

🕮 3.11.2

String literals

You can include predefined string values in your code as string literals. A string literal is a fixed sequence of text characters surrounded by a pair of quotation marks (""). Use a string literal as the initial value for a constant or variable:

```
let someString = "Some string literal value"
```

Note that Swift derives a string type for the someString constant because it is initialized with a string literal value.

🕮 3.11.3

Multiline string literals

If you need a string that includes several lines, use a multiline string literal - a sequence of characters surrounded by three quotation marks:

```
let text = """
```

Some multiline text,

which can contain line breaks.

```
"""
```

A multiline string literal contains all the lines between its opening and closing quotation marks. The string begins on the first line after the opening quotation marks (""") and ends on the line before the closing quotation marks, which means that none of the following strings begin or end with a line break:

```
let singleLineString = "They are the same."
let multilineString = """
They are the same.
"""
```

When your source code contains a line break inside a multiline string literal, that line break also appears in the string value. To use line breaks to make the source code easier to read, but you don't want line breaks to be part of the string value, type a backslash (\) at the end of these lines:

```
let text = """
```

Some multiline text, \

which can contain line breaks.

```
"""
```

3.11.4

What are the three quotation marks """ for?

- Causes a compilation error
- To insert special (diacritical) signs
- For inserting multiline texts

📖 3.11.5

Special characters in text literals

Text literals can contain the following special characters:

Escaped characters

\0 (zero), \\(backslash), \t (tab), \n (newline), \r (return), \"(quotation mark), and \'(apostrophe)

Any Unicode scalar written as \u{n}, where n is a 1-8 digit hexadecimal number with a value equal to a valid Unicode code

```
let wiseWords = "\"Imagination is more important than
knowledge \"- Einstein"
// prints "Imagination is more important than knowledge" -
Einstein
let dollarSign = "\u{24}"
// $, Unicode scalar character U + 0024
let blackHeart = "\u{2665}" // ♥, Unicode U + 2665 scalar
character
```

📖 3.11.6

Extended string delimiters

You can place a string literal in extended delimiters to include special characters in the string without causing them to take effect. Place the string in quotation marks (") and surround it with a pound sign (#). When you need to type a special character, add the # character between the backslash and the character:

```
print (# “line1\nline2” #)
// prints line1\nline2
print(# “line1\#nline2” #)
// prints line1
// line2
```

📖 3.11.7

Initialization of an empty string

To create an empty string value as the default value for creating a longer string, assign an empty string literal to the variable or initialize a new instance of the string using the initializer syntax:

```
var emptyString = ""
// empty string literal
var anotherEmptyString = String()
// initializer syntax
// these two strings are empty and are equivalent to each
other
```

Use the isEmpty property (the return value is of type Boolean) to determine if the string is empty:

```
if emptyString.isEmpty {
  print("Nothing to see here")
}
// prints "Nothing to see here"
```

3.11.8

Changes in Strings

You can choose whether or not to edit a particular string. When you create a variable, the string will change. When you create a constant, the string will be immutable:

```
var variableString = "Horse"
variableString += "a carriage"
// variableString is now "Horse and Carriage"
let constantString = "Mountaineer"
constantString += "and another mountaineer"
// this reports a compilation error - the constant string
cannot be modified
```

3.11.9

Can the += operator be applied to a string?

- yes
- no

3.12 Working with strings and characters

🕮 3.12.1

String values

If you create a new value of type String, that value is copied when it is passed to a function or method or when it is assigned to a constant or variable. A new copy of the existing string value is always created, and a new copy is passed or assigned.

This copy behavior ensures that when a function or method passes a String value to you, it is clear that you own that exact String value, regardless of where it comes from. You can count on the string passed to you not being modified unless you change it yourself.

🕮 3.12.2

Working with characters

Swift String represents a collection of character values in a specified order. You can access individual characters in a string in a loop:

```
for character in “Dog!???? “{
  print(character)
}
// D
// o
// g
//!
// ????
```

You can create a separate character constant or variable from a single-character string literal:

```
let exclamationMark: Character = "!"
```

String values can be constructed by passing an array of characters to its initializer:

```
let catCharacters: [Character] = ["C", "a", "t", "!", "????"]
let catString = String (catCharacters)
print (catString)
// prints “Cat!???? “
```

3.12.3

What type does the character represent

- Char
- char
- Character
- character

3.12.4

Concatenation of texts and characters

String values can be concatenated (added) with an addition (+) operator to create a new string value:

```
let string1 = "hello"
let string2 = "everyone"
var welcome = string1 + string2
// The welcome variable contains the value "hello everyone"
```

You can also add a String value to an existing String variable using the add assignment operator (+=):

```
var instruction = "look around"
instruction += string2
// the instruction now equals "look around everyone"
```

You can add a Character value to a String variable using the String append() method:

```
let exclamationMark: Character = "!"
welcome.append(exclamationMark)
// The welcome variable contains the value "hello everyone!"
```

You cannot append a string or character to an existing Character variable because the Character value must contain only one character.

3.12.5

String interpolation

String interpolation is a way to create a new string value from a mixture of constants, variables, literals, and expressions by including their values in the string literal. Wrap each item that you put in a string literal in a pair of parentheses with a backslash prefix:

```
let multiplier = 3
let message = "\(multiplier) times 2.5 is \(Double(multiplier)
* 2.5)"
// message is "3 times 2.5 is 7.5"
```

In the example above, the multiplier value is inserted into the string literal as \(multiplier). This placeholder is replaced by the actual value of the multiplier when the string interpolation is evaluated when the actual string is created.

The multiplier value is also part of another expression further down the string. This expression calculates the value Double (multiplier) * 2.5 and inserts the result (7.5) into a string.

3.12.6

How do you insert the value of a numeric variable a into a string?

- "Value of a is \(a)"
- "Value of a is " + a
- "Value of a is " + str(a)

3.12.7

Extended clusters of graphemes

Each Character instance represents one extended cluster of graphemes — a sequence of one or more Unicodes that (when combined) create a single human-readable character.

For example, the letter é can be expressed as é Unicode (U+00E9). However, the same letter can also be represented as a pair of codes - the standard letter e (or U+0065), followed by the diacritics code (U+0301). This code is graphically applied to the code that precedes it and turns e into é.

In both cases, the letter é is represented as a single Character value, which represents an extended cluster of graphemes. In the first case, the cluster contains one code; in the second case it is a cluster of two codes:

```
let eAcute: Character = "\u{E9}" // é
let combineEAcute: Character = "\u{65}\u{301}" // e followed
by '
// eAcute is é, combineEAcute is é
```

📖 3.12.8

Character counting

To know the number of characters in a string, call the count property of the string:

```
let text = "Hello everyone"
print("text has \(text.count) characters")
// prints "text has 12 characters
```

If you initialize a new string with the word cafe and then append it to the end of the string (U+0301), the resulting string will still have 4 characters:

```
var word = "cafe"
print("number of characters in \(word) is \(word.count)")
// prints "number of characters in cafe is 4"
word += "\u{301}"
print("number of characters in \(word) is \(word.count)")
// prints "the number of characters in cafe is 4
```

📝 3.12.9

How do we find the number of characters in the variable s?

- s.countOfChars
- s.length
- s.count

📖 3.12.10

String access and modification

Access and modify the string using its methods and properties or the index syntax.

String indexes

Each string value has an associated index type, String.Index, that corresponds to the position of each character in the string.

Use the startIndex property to access the position of the first character of the string. The endIndex property is the position after the last character in the string. As a result, the endIndex property is not a valid argument for the string index. If the string is empty, startIndex and endIndex are the same.

You access indexes before and after a given index using the index (before:) and index (after:) string methods. You can also use the index (_:offsetBy:) method to move the index instead of calling one of these methods multiple times.

You can use index syntax to access a character in a particular string index.

```
let greeting = "Good day!"
greeting[greeting.startIndex]
// G
greeting[greeting.index(before: greeting.endIndex)]
//!
greeting[greeting.index(after: greeting.startIndex)]
// u
let index = greeting.index(greeting.startIndex, offsetBy: 7)
greeting[index]
// a
```

Attempting to access an out-of-string index or a character in an out-of-string index will cause a run-time error.

```
greeting[greeting.endIndex] // Error
greeting.index (after: greeting.endIndex) // Error
```

Use the indices property to access all the indexes of each character in the string.

```
for index in greeting.indices {
  print("\(greeting[index])", terminator: "")
}
// prints "G u t e n T a g!"
```

3.13 Insertion into substrings and characters

🕮 3.13.1

Inserting into a string

To insert one character into a string at a given index, use the insert(_: at:) method, and to insert the contents of another string into a given index, use the insert (contentsOf: at:) method:

```
var welcome = "hello"
welcome.insert("!", at: welcome.endIndex)
// welcome contains "hello!"
welcome.insert(contentsOf: " there", at: welcome.index(before:
welcome.endIndex))
// welcome contains "hello there!"
```

3.13.2

Which command do you use to insert a substring in a string?

- insert(“Hello”, at:s.endIndex)
- insert(contentsOf:”Hello”, at:s.endIndex)
- insert(contentsOf:”Hello”, s.endIndex)

3.13.3

Removing from string

To remove a single character from a string at a given index, use the remove(at :) method, and to remove a substring at a given index, use the removeSubrange(_ :) method:

```
welcome.remove(at: welcome.index(before: welcome.endIndex))
// welcome contains "hello there"
let range = welcome.index(welcome.endIndex, offsetBy: -
6)..<welcome.endIndex
welcome.removeSubrange(range)
// welcome contains "hello"
```

3.13.4

Substrings

When you retrieve a substring from a string — for example, using the index or the prefix(_ :) method — the result is an instance of the substring, not another string. Substrings in Swift have most of the same methods as strings, which means that you can work with substrings in the same way as strings. Substrings use the memory of the strings from which they were created. When you need to save the result for a long time, convert the substring to a string instance:

```
let greeting = "Hello, world!"
let index = greeting.firstIndex(of: ",") ?? greeting.endIndex
let beginning = greeting[..<index]
// beginning is "Hello"
// Convert the result to a String for long-term storage.
let newString = String(beginning)
```

3.13.5

Is it recommended to use substrings to store string values in the long run?

- yes
- no

3.13.6

Equality of strings and characters

The equality of strings and characters is checked using the "equals" operator (==) and the "does not equal" (!=) operator, as described in the comparison operators:

```
let quotation = "We are very similar, you and me."
sameQuotation = "We are very similar, you and me."
if quotation == sameQuotation {
  print("These two strings are considered equivalent")
}
// prints "These two strings are considered equivalent
```

Two string values (or two character values) are considered equivalent if their extended graphema clusters are canonically equivalent. Extended graph cluster clusters are canonically equivalent if they have the same linguistic meaning and appearance, even though they are composed of different Unicode scalars behind the scenes.

3.13.7

Can comparison operators (==,!=) Be used with variables of type String?

- yes
- no

3.13.8

Equality of the beginning and end of the string

To check whether a string begins or ends with specific text, call the hasPrefix(_:) and hasSuffix(_:) methods of the string, both of which have a single String argument and return a Boolean value:

```
var text = "Today was nice weather."
if text.hasPrefix(“Today was”) {
  print("text starts the same")
}
// prints "text starts the same"
if text.hasSuffix(“weather”) {
   print("text ends the same")
}
// It doesn't print anything because it ends with "weather."
and not "weather"
```

3.13.9

Unicode string representations

When a Unicode string is written to a text file or other storage, the Unicode codes in that string are encoded into one of several encoding forms defined by Unicode. Each form encodes a string in small blocks, code units. These include UTF-8 encoding (8-bit code units), UTF-16 (16-bit code units), and UTF-32 (32-bit code units).

UTF8

```
let dogString = "Dog!!????"
"For codeUnit in dogString.utf8 {
  print ("\(codeUnit)", terminator: "")
}
print ("")
// prints "68 111 103 226 128 188 240 159 144 182"
```

UTF16

```
let dogString = "Dog!!????"
"For codeUnit in dogString.utf16 {
  print ("\(codeUnit)", terminator: "")
}
print ("")
```

```
// Prints "68 111 103 8252 55357 56374"
```

UNICODE

```
let dogString = "Dog‼????"
„For scalar in dogString.unicodeScalars {
  print ("\(scalar.value)", terminator: "")
}
print ("")
// Prints "68 111 103 8252 128054"
```

Collections

Chapter 4

4.1 Collections

📖 4.1.1

Collections

Swift provides three types of collections for storing value collections: arrays, sets, and dictionaries. Arrays are ordered collections of values. Sets are disordered collections of unique values. Dictionaries are disordered collections of key-value pairs. Arrays, sets, and dictionaries always have clearly defined types of values and keys in Swift that they can store. This means that you can't accidentally insert an incorrect type value into a collection, and you can be sure of the type of values you retrieve from the collection.

📖 4.1.2

Editing collections

If you create an array, set, or dictionary and assign it to a variable, the created collection will be changeable. This means that once created, you can change the collection by adding, removing, or changing items in the collection. If you assign an array, set, or dictionary to a constant, the collection is immutable and its size and contents cannot be changed.

📖 4.1.3

Arrays

The arrays stores values of the same type in a sorted list. The same value can appear several times in the array in different positions.

Array type syntax:

```
Array[Element]
Array<Element>
```

where Element is the type of values that the array is allowed to store. These two syntaxes are functionally identical, it is preferable to use the syntax with square brackets [Element].

Arrays in Swift are always indexed from 0.

🕮 4.1.4

Creating an empty array

You can create an empty array of a certain type using the initializer syntax:

```
var someInts = [Int]()
print("someInts is of type [Int] with \(someInts.count).")
// prints "someInts is of type [Int] with 0 entries."
```

Note that the type of the someInts variable is derived as [Int] from the initializer type. Alternatively, if the context already provides type information, such as a function argument or a variable or constant already specified, you can create an empty array with an empty array literal that is written as [] (an empty pair of square brackets):

```
someInts.append(3)
// someInts now contains 1 value of type Int
someInts = []
// someInts is now an empty array, but still of type [Int]
```

🕮 4.1.5

Create an array with a default value

The Array type provides an initializer for creating an array of a certain size with all its values set to the same default value. This initializer expects two parameters (repeating with the default value and count with the number of values in the array):

```
var threeDoubles = Array(repeating: 0.0, count: 3)
// threeDoubles is of type [Double] and equals [0.0, 0.0, 0.0]
```

🕮 4.1.6

Create an array by adding two arrays

You can create a new array by adding two existing arrays with compatible types using the add (+) operator. The type of the new array is derived from the type of the two arrays you are joining:

```
var anotherThreeDoubles = Array (repeating: 2.5, count: 3)
// anotherThreeDoubles is of type [Double] and equals [2.5,
2.5, 2.5]
var sixDoubles = threeDoubles + anotherThreeDoubles
```

```
// sixDoubles is derived as [Double] and equals [0.0, 0.0,
0.0, 2.5, 2.5, 2.5]
```

4.1.7

Create an array with values

You can also initialize an array using an array literal, which is an abbreviation for writing one or more values as an array collection. The array literal is written as a comma-separated list of values, surrounded by a pair of square brackets:

```
var shoppingList: [String] = ["Eggs", "Milk"]
// shoppingList was initialized with two initial items
```

The shoppingList variable is declared as an "array of string values" [String]. Because this particular array specified the "String" value type, only String values are allowed to be stored.

Because all values in the array literal are of the same type, we can also initialize the array as follows:

```
var shoppingList = ["Eggs", "Milk"]
```

4.1.8

What literal is used to initialize an array?

4.1.9

Access to array elements

You can access and edit an array using methods and properties or using index syntax.

The number of items in the array

To determine the number of items in an array, check its read-only count property:

```
print("The shopping list contains \(shoppingList.count)
items.")
// prints "Shopping list contains 2 items."
```

Use the isEmpty Boolean property to check that the count property is 0:

```
if shoppingList.isEmpty {
  print("The shopping list is empty.")
} else {
  print("The shopping list is not empty.")
}
// prints "Shopping list is not empty."
```

4.1.10

Add an item to the end of an array

You can add a new entry to the end of an array by calling the append (_:) array method:

```
shoppingList.append("Flour")
// shoppingList now contains 3 items
```

You can use the add (+=) operator to add a compatible item (s) to a field:

```
shoppingList += ["Baking Powder"]
// shoppingList now contains 4 items
shoppingList += ["Chocolate Spread", "Cheese", "Butter"]
// shoppingList now contains 7 items
```

4.1.11

Can I use the += operator to add items to a field?

- yes
- no

4.1.12

Access array values using an index

You can read a value from an array using the index in square brackets immediately after the array name:

```
var firstItem = shoppingList[0]
// firstItem equals "Eggs"
```

You can use an index to change existing values at that index:

```
ShoppingList[0] = "Six Eggs"
// the first item in the list now equals "Six Eggs"
```

When using an index, the specified index must be valid. For example, writing shoppingList[shoppingList.count] = "Salt" that attempts to append an item to the end of an array will cause a runtime error.

4.1.13

Changing array values

You can also use subscript syntax to change a range of values at once, even if the replacement set of values has a different length than the range you are replacing. The following example replaces the terms "Chocolate spread", "Cheese" and "Butter" with "Bananas" and "Apples":

```
shoppingList[4...6] = ["Bananas", "Apples"]
// shoppingList now contains 6 items
```

4.1.14

Which method is used to insert a value on index 2?

- insert(_: at:)
- insert(index: value:)
- insert(_: value:)

4.1.15

Delete an item from the array

To remove an item from the array, call the remove(at :) method. This method deletes an entry in the specified index and returns "the deleted entry (although you can ignore it if you don't need it):

```
let mapleSyrup = shoppingList.remove(at: 0)
// the item that was at index 0 has just been deleted
// shoppingList now contains 6 items and no maple syrup
// the mapleSyrup constant now equals the removed "Maple
Syrup" string
```

When an item is removed from an array, a space in the array is dropped. Now the value in index 0 is again equal to "Six eggs".

```
firstItem = shoppingList [0]
// firstItem now equals "Six Eggs"
```

To remove the last item from an array, use the removeLast() method instead of the remove(at :) method to avoid having to query the count property of the array. Like the remove(at :) method, removeLast() returns the removed item:

```
let apples = shoppingList.removeLast()
// the last item in the array has just been deleted
// shoppingList now contains 5 items and no apples
```

🕮 4.1.16

Insert a value into the array at the given index

To insert an entry into an array at the specified index, call the insert(_: at :) method:

```
ShoppingList.insert("Maple syrup", at: 0)
// shoppingList now contains 7 items
// "Maple syrup" is now the first item in the list
```

Calling the insert(_: at :) method inserts a new item with the value "Maple Syrup" at the beginning of the shopping list under index 0.

🕮 4.1.17

Iteration over the array

You can traverse the entire array using a for-in loop:

```
for item in shoppingList {
  print(item)
}
// Six eggs
// Milk
// Flour
// Baking powder
// Bananas
```

If you need an integer index of each item and its value, use an iteration through the enumerated() method instead, which returns an integer (index) and an item for each item in the array. The index starts at zero and rises by one for each item:

```
for (index, value) in shoppingList.enumerated () {
  print("Item \(index + 1): \(value)")
}
// Item 1: Six eggs
// Item 2: Milk
// Item 3: Flour
// Item 4: Baking powder
// Item 5: Bananas
```

4.1.18

What value does the enumerated method return?

- array item
- index of the array
- n-tuple (index, value)

4.2 Sets

4.2.1

Sets

The set stores different values of the same type in a collection without a defined sort order. If the order of the items is not important, or if you need to ensure that the item appears only once, you can use a set instead of an array.

Set syntax

The Swift set type is written as Set<Element>, where Element is the type that the set is allowed to store.

4.2.2

Create and initialize an empty set

You can create an empty set of a specific type using the initializer syntax:

```
var letters = Set<Character>()
print ("Set<Character> letters with \(letters.count)
entries.")
// prints “Set <Character> letters with 0 entries.
```

4.2.3

Create an array-initialized set

You can also initialize a set with array literal, as an abbreviated way to write one or more values as a set collection.

The following example creates a set called favoriteGenres to store string values:

```
var favoriteGenres: Set<String> = ["Rock", "Classical", "Hip
hop"]
// favorites Genres were initialized with three starting items
```

The favoriteGenres variable is declared as a "set of string values", written as Set<String>. Because this particular set has specified a String value type, only String values are allowed to be stored.

The set type cannot be derived from the array literal itself, so the Set type must be explicitly declared. However, to derive an item type, you do not have to write the element type of the set if you initialize it with an array literal that contains values of only one type:

```
var favoriteGenres: Set = ["Rock", "Classical", "Hip hop"]
```

4.2.4

Which command creates a new set?

- var favoriteGenres: Set<|String> = ["Rock", "Classical", "Hip hop"]
- var favoriteGenres: Set(String) = ["Rock", "Classical", "Hip hop"]
- var favoriteGenres: Set[String] = ["Rock", "Classical", "Hip hop"]

4.2.5

Access to set values

You access a set using its methods and properties.

To find out the number of items in a set, check its read-only count property:

```
print ("I have \(favoriteGenres.count) favorite music
genres.")
// prints "I have 3 favorite music genres."
```

Use the isEmpty Boolean property to check that the count property is 0:

```
if favoriteGenres.isEmpty {
  print("I'm not picky about music.")
} else {
  print("I have specific music preferences.")
}
// prints "I have certain music presets."
```

To check if a set contains a specific item, use the contains(_ :) method:

```
if favoriteGenres.contains("Funk") {
  print("Funk is my favorite style.")
} else {
  print("I'm not listening to Funk.")
}
// prints "I'm not listening to Funk.
```

📖 4.2.6

Changes to set values

You modify the values of a set using its methods and properties.

You can add a new item to the set by calling the insert(_:) method:

```
favouriteGenres.insert("Jazz")
// Favorite Genres now contains 4 items
```

You can remove an item from a set by calling the remove(_:) method, which removes the item if it is a member of the set and returns the removed value, or returns nil if the set did not contain it. You can also remove all items in a set using the removeAll() method.

```
if let removeGenre = favoriteGenres.remove ("Rock") {
  print ("\(removeGenre)? I'm not listening to it anymore.")
} else {
  print("I never cared much about it.")
}
```

```
// prints "Rock? I'm not listening to it anymore. "
```

4.2.7

What method do you use to insert a new element into a set?

- insert()
- insert(_:)
- insert(value:)

4.2.8

Iteration over the set

You can iterate over the values in a set using a for-in loop:

```
for genre in favoriteGenres {
  print("\(genre)")
}
// Classical
// Jazz
// Hip hop
```

Sets do not have a defined sort order. To iterate over the sorted values of a set, use the sorted () method:

```
for genre in favoriteGenres.sorted () {
  print("\(genre)")
}
// Classical
// Hip hop
// Jazz
```

4.2.9

Basic operations with sets

You can efficiently perform basic operations on sets, such as combining two sets together, determining which values have two sets in common, or determining whether two sets contain all, some, or none of the same values.

- Use the intersection(_:) method to create a new set only with values common to both sets.

- Use the symmetricDifference(_:) method to create a new set with values that are only in one set.
- Use the union(_:) method to create a new set with all the values that are in both sets.
- Use the subtracting(_:) method to create a new set with values that are not in the specified set.

```
let oddDigits: Set = [1, 3, 5, 7, 9]
let evenDigits: Set = [0, 2, 4, 6, 8]
let singleDigitPrimeNumbers: Set = [2, 3, 5, 7]
oddDigits.union(evenDigits).sorted ()
// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
oddDigits.intersection(evenDigits).sorted ()
// []
oddDigits.subtracting(singleDigitPrimeNumbers).sorted ()
// [1, 9]
oddDigits.symmetricDifference(singleDigitPrimeNumbers).sorted
()
// [1, 2, 9]
```

4.2.10

Compare sets

Use the equals (==) operator to specify whether the two sets contain all the same values.

Use the isSubset(of:) method to determine if all set values are included in the specified set.

Use the isSuperset(of:) method to determine if the set contains all the values in the specified set.

You can use the isStrictSubset(of:) or isStrictSuperset(of:) methods to determine whether a set is a subset or a superset, but is not equal to the specified set.

Use the isDisjoint(with:) method to determine if the two sets have common values.

```
let houseAnimals: Set = ["????", "????"]
let farmAnimals: Set = ["????", "????", "????", "????",
"????"]
let cityAnimals: Set = ["????", "????"]
houseAnimals.isSubset(of: farmAnimals)
// true
farmAnimals.isSuperset(of: houseAnimals)
// true
```

```
farmAnimals.isDisjoint(with: cityAnimals)
// true "
```

4.2.11

Can you use the == operator to compare two sets?

- yes
- no

4.3 Dictionaries

4.3.1

Dictionaries

The dictionary stores key-value pairs. All keys in the dictionary must be of the same type. All values in the dictionary must be of the same type. Each value is associated with a unique key that acts as an identifier for that value in the dictionary. Unlike array entries, dictionary entries do not have a specified order. You use the dictionary when you need to look up values based on their identifier.

Dictionary syntax

The dictionary type is defined as:

```
Dictionary[Key: Value]
Dictionary<Key, Value>
```

Key is a type of value that can be used as a key, and Value is a type of value. These two shapes are functionally identical, the form with square brackets Dictionary [Key: Value] is preferred.

4.3.2

Create an empty dictionary

As with arrays, you can create an empty dictionary of a certain type using the initializer syntax:

```
var namesOfIntegers = [Int: String]()
// namesOfIntegers is an empty dictionary [Int: String]
```

In this example, an empty dictionary of type [Int: String] is created, in which readable names of integer values will be stored. Its keys are of type Int and its values are of type String. If the context already provides type information, you can create an empty dictionary with an empty dictionary literal written as [:] (a colon inside the double square brackets):

```
namesOfIntegers[16] = "sixteen"
// namesOfIntegers now contains 1 key-value pair
namesOfIntegers = [:]
// namesOfIntegers is again an empty dictionary of type [Int:
String]
```

4.3.3

What initializer creates an empty dictionary with an Int key and a Double value?

- [Int: Double]
- [Int: Double]()
- [Int: Double][]

4.3.4

Creating a dictionary using a literal

You can also initialize a dictionary using a literal that has a similar syntax to an array literal. A dictionary literal is an abbreviated way of writing one or more key-value pairs as a dictionary collection. In the literal dictionary, the key and value in each pair are separated by a colon. Pairs are written as a comma-separated list, surrounded by a pair of square brackets:

```
[key1: value1, key2: value2, key3: value3]
```

The example below creates a dictionary for storing international airport names. In this dictionary, the keys are the three-letter codes of the International Air Transport Association and the values are the names of the airports:

```
var airports: [String: String] = ["YYZ": "Toronto Pearson",
"DUB": "Dublin"]
```

The airport dictionary is declared as type [String: String], which means “dictionary with String type keys and String type values.

As with arrays, you don't have to define a dictionary type if you initialize it with a dictionary literal whose keys and values have clear types:

```
var airports = ["YYZ": "Toronto Pearson", "DUB": "Dublin"]
```

4.3.5

What literal is used to define the dictionary?

- [ke1, ke2: value1, value2]
- [key1: value1, key2: value2]
- [key1, value1, key2, value2]
- [value1: key1, value2: value2]

4.3.6

Access to the dictionary

You can access the dictionary using methods and properties or through an index.

As with arrays, you can find out the number of entries in the dictionary by checking its read-only count property:

```
print("The airport dictionary contains the entries
\(airports.count).")
// prints "Airport dictionary contains 2 entries."
```

Use the boolean property isEmpty to check if the count property is 0:

```
if airports.isEmpty {
  print("The airport dictionary is empty.")
} else {
  print("Airport dictionary is not empty.")
}
// prints "The airport dictionary is not empty."
```

You can use the index to read dictionary values for a specific key. Because it is possible to request a key for which there is no value, the dictionary index returns an optional value for the dictionary value type. If the dictionary contains a value for the required key, the index returns an optional value containing the existing value for that key. Otherwise, the subscript returns nil:

```
if let airportName = airports["DUB"] {
  print("The airport name is \(airportName).")
} else {
  print("This airport is not in the airport dictionary.")
}
```

```
// prints "The airport name is Dublin.
```

4.3.7

How do you read a value from the pair in a dictionary?

- value = dictionary["key"]
- value = dictionary.key
- value = dictionary.value["key"]

4.3.8

Dictionary editing

You can edit the dictionary using methods and properties or using an index.

You can add a new index entry to the dictionary. Use the new key of the appropriate type as the index and assign a new value of the appropriate type:

```
airports ["LHR"] = "London"
// the airport dictionary now contains 3 entries
```

You can also use an index to change the value associated with a particular key:

```
airports["LHR"] = "London Heathrow"
// the value for "LHR" has been changed to "London Heathrow"
```

As an alternative to index access, use the updateValue(_: forKey:) method to set or update a value for a specific key. However, unlike index access, the updateValue(_: forKey:) method returns the old value after the update. This will allow you to check if the update has taken place.

```
if let oldValue = airports.updateValue("Dublin Airport",
forKey:"DUB") {
  print("The old value for DUB was \(oldValue).")
}
// prints "The old value for DUB was Dublin."
```

📖 4.3.9

Deleting in the dictionary

You can use an index to remove a key-value pair from the dictionary by assigning a nil value to that key:

```
airports["APL"] = "Apple International"
// "Apple International" is not a real airport for APL, so
delete it
airports["APL"] = nil
// APL has now been removed from the dictionary
```

Or you can remove a pair with the removeValue (forKey :) method. This method deletes the pair if it exists and returns the deleted value, or returns nil if no value existed:

```
if let removedValue = airports.removeValue(forKey: "DUB") {
  print("The name of the removed airport is \(removedValue).")
} else {
  print("The airport dictionary does not contain a value for
DUB.")
}
// prints "The name of the removed airport is Dublin Airport."
```

📝 4.3.10

What method do you use to remove a pair in the dictionary?

- removePair()
- removePair(forKey:)
- removeValue()
- removeValue(forKey:)

📖 4.3.11

Iteration over the dictionary

Iterate over the dictionary using a for-in loop. Each entry in the dictionary is returned as an n-tuple (key, value), and as part of an iteration you can decompose the n-tuple into temporary constants or variables:

```
for (airportCode, airportName) in airports {
  print ("\(airportCode): \(airportName)")
}
// LHR: London Heathrow
// YYZ: Toronto Pearson
```

Or you can iterate over an array of keys or values by accessing the keys and values properties:

```
for airportCode in airports.keys {
  print ("Airport code: \(airportCode)")
}
// Airport code: LHR
// Airport code: YYZ
for airportName in airports.values {
  print ("Airport name: \(airportName)")
}
// Airport name: London Heathrow
// Airport name: Toronto Pearson
```

If you need to use dictionary keys or values as an Array, initialize the new array using the keys or values property:

```
flight airportCodes = [String](airports.keys)
// airportCodes is ["LHR", "YYZ"]
flight airportNames = [String](airports.values)
// airportNames is ["London Heathrow", "Toronto Pearson"]
```

The dictionary is not sorted. To iterate over sorted keys or dictionary values, use the sorted() method on its keys or values properties.

Functions

Chapter 5

5.1 Functions and work with them

5.1.1

Functions

Functions are separate blocks of code that perform a specific task. A function has a name that identifies what it is doing, and that name is used to "call" the function.

The syntax of a Swift function is flexible enough to express anything from a simple C-style function without parameter names to a complex Objective-C method with names and argument names for each parameter. Parameters can provide default values to simplify function calls. Parameters can be passed by reference so that they modify the value of a parameter when the function finishes.

Each function in Swift has a type, which consists of function parameter types and a return type. You can use this type like any other type in Swift, which makes it easier to pass functions as parameters to other functions and return functions from functions. Functions can also be written to other functions to encapsulate useful functions within a range of nested functions.

5.1.2

Simple function

The function in the example below is called greet(person :). It takes the person's name as input and returns a greeting to that person. To do this, you define one input parameter - a string value called person - and a String return type that will contain a greeting for that person:

```
func greet(person: String) -> String {
  let greeting = "Hello," + person + "!"
  return greeting
}
print(greet(person: "Anna"))
// prints "Hello, Anna!"
print(greet(person: "Brian"))
// prints "Hi, Brian!"
```

5.1.3

Which function header is correct?

- func test(a: Int) -> Int

- function test(a: Int): Int
- func test(a: Int): Int

🕮 5.1.4

Functions with multiple parameters

Functions can have multiple input parameters, which are written in definition parentheses, separated by commas.

This function takes the person's name and whether it has already been welcomed and returns the corresponding greeting to that person:

```
func greet (person: String, alreadyGreeted: Bool) -> String {
  if alreadyGreeted {
    return "Hi," + person + "!"
  } else {
    return "Hello again," + person + "!"
  }
}
print(greet(person: "Tim", alreadyGreeted: true))
// prints "Hello again, Tim!"
```

🕮 5.1.5

Functions without parameters

Functions do not have mandatory input parameters. Here is a function without input parameters that always returns the same text value on each call:

```
func sayHelloWorld () -> String {
  return "Hello, world"
}
print(sayHelloWorld())
// prints "Hello, world"
```

The function definition requires parentheses after the function name, even if it does not accept any parameters. An empty pair of parentheses is also required when calling a function.

5.1.6

Which function header is correct?

- function test(a: Int; b: String): Int
- func test(a: Int, b: String) -> Int
- func test(a: Int, b: String) -> Int

5.1.7

Functions without return value

Functions do not have to define a return value. Here is the version of the greet (person:) function that prints the text directly instead of returning it. The definition of such a function will contain neither the arrow -> nor the type of return value:

```
func greet (person: String) {
  print("Hi, \(person)!")
}
greet (person: "Dave")
// prints "Hi, Dave!"
```

5.1.8

Ignoring the return value

The return value of the function can be ignored, as shown below:

```
func printAndCount(string: String) -> Int {
  print(string)
  return string.count
}
func printWithoutCounting(string: String) {
  let _ = printAndCount(string: string)
}
printAndCount(string: "hello world")
// prints "hello world" and returns 10
printWithoutCounting(string: "hello world")
// prints "hello world" but does not return a value
```

5.1.9

Is it possible to define a function in Swift that returns multiple values at once?

- No, Swift doesn't allow anything like that
- Yes, but only an Array value
- Yes, for example by returning an N-tuple type

5.1.10

Optional return n-tuple types

The above minMax (array:) function does not perform any security checks on the array passed to it. If the array argument contains an empty array, the minMax (array:) function raises a runtime error when attempting to access array [0]. If an empty n-tuple can be returned by the function, use the optional n-tuple type by placing a question mark after the closing bracket of the n-tuple type, for example (Int, Int)? or (String, Int, Bool)?:

```
func minMax(array: [Int]) -> (min: int, max: int)? {
  if array.isEmpty {
    return nil
  }
  var currentMin = array[0]
  var currentMax = array[0]
  for value inarray[1..<array.count] {
    if value < currentMin {
      currentMin = value
    } else if value > currentMax {
      currentMax = value
    }
  }
  return (currentMin, currentMax)
}
if let bounds = minMax (pole: [8, -6, 2, 109, 3, 71]) {
  print ("min is \(bounds.min) and max is \(bounds.max)")
}
// prints "min is -6 and max is 109"
```

🕮 5.1.11

Functions with multiple return values

As a function return type, you can use the n-tuple type to return multiple values as part of a single compound return value. The following example defines a function called minMax(array:), which finds the smallest and largest number in an array of Int values and returns both:

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
  var currentMin = array[0]
  var currentMax = array[0]
  for value in array[1..<array.count] {
    if value < currentMin {
      currentMin = value
    } else if value > currentMax {
      currentMax = value
    }
  }
  return (currentMin, currentMax)
}
```

Because n-tuple member values are named as part of the return type of a function, they can be accessed using dot syntax to retrieve the minimum and maximum values found, and the names do not need to be defined in the return statement:

```
let bounds = minMax(array: [8, -6, 2, 109, 3, 71])
print ("min is \(bounds.min) and max is \(bounds.max)")
// prints "min is -6 and max is 109"
```

🕮 5.1.12

Function with default return

If the whole body of a function is a single expression, the function implicitly returns that expression without having to call return. For example, both of the following functions have the same behavior:

```
func greeting(for person: String) -> String {
  "Hi," + person + "!"
}
print(greeting(for: "Dave"))
// It says "Hi, Dave!"
func anotherGreeting(for person: String) -> String {
  return "Hi," + person + "!"
}
print(anotherGreeting(for: "Dave"))
// prints "Hi, Dave!
```

5.1.13

Is the following function defined correctly?

```
func returnDouble(number: Int) -> Int {
  number * 2
}
```

- yes
- no

5.1.14

Labels and names of parameters passed to the function

Each function parameter has a label and a parameter name. The parameter label is used when calling a function. The parameter name is used when implementing the function. By default, functions use the same labels and parameter names.

```
func someFunction(firstParameterName: Int,
secondParameterName: Int) {
  // In the body of the firstParameterName and
secondParameterName functions
  // reference to the argument values for the first and second
parameters.
}
someFunction(firstParameterName: 1, secondParameterName: 2)
```

All parameters must have unique names. Although it is possible for multiple parameters to have the same argument label, unique argument labels can help you read your code.

5.2 Labels, parameters

🕮 5.2.1

Other parameter labels

Before the name of the parameter, write its label, separated by a space:

```
func someFunction (argumentLabel parameterName: Int) {
  // In the body of the function, parameterName refers to the
value of the argument
  // for this parameter.
}
```

Example:

```
func greet (person: String, from hometown: String) -> String {
  return "Hi \(person)! I'm glad you were able to visit
\(hometown)."
}
print (greet(person: "Bill", from: "Cupertino"))
// prints "Hi Bill! I'm glad you were able to visit Cupertino.
"
```

🕮 5.2.2

Omitting the label

If you do not want a label for the parameter, type an underscore (_) instead of a label:

```
func someFunction (_ firstParameterName: Int,
secondParameterName: Int) {
  // In the body of the firstParameterName and
secondParameterName functions
  // reference to the argument values for the first and second
parameters.
}
someFunction (1, secondParameterName: 2)
```

5.2.3

What character in the function definition can we omit the parameter label?

- * (asterisk)
- _ (underscore)
- - (dash)

5.2.4

Default parameter values

You can define a default value for any parameter in a function by assigning a parameter value to the type of that parameter. If a default value is defined, you can omit this parameter when calling the function. Place non-default parameters at the top of the function parameter list before parameters that have default values. The readability of your code will be better:

```
func someFunction(parameterWithoutDefault: Int,
parameterWithDefault: Int = 12) {
  // If you omit the second argument when calling this
function, then
  // value parameterWithoutDefault is 12 inside the body of
the function.
}
someFunction(parameterWithoutDefault: 3, parameterWithDefault:
6)
// parameterWithDefault is 6
someFunction(parameterWithoutDefault: 4)
// parameterWithDefault is 12
```

5.2.5

How do we write the default value of a parameter in the function definition?

- func test(a: Int = 10)
- func test(a = 10: Int)
- func test(10 = a: Int)

5.2.6

Variadic parameter

The variadic parameter accepts none, one, or more values of the specified type. You use a varied parameter to pass a different number of parameter values when calling a function. You write the VIC parameters with three periods (...) after the name of the parameter type. The function can have a maximum of one variation parameter.

The values of the variadic parameter are made available in the body of the function as a field of the appropriate type:

```
func arithmeticMean(_ numbers: Double ...) -> Double {
  var total: Double = 0
  for number in numbers {
    total += number
  }
  return total / Double(numbers.count)
}
arithmeticMean (1, 2, 3, 4, 5)
// returns 3.0, which is the arithmetic mean of these five
numbers
arithmeticMean (3, 8.25, 18.75)
// returns 10.0, which is the arithmetic mean of these three
numbers
```

5.2.7

What type is the variadic parameter (...) of the function represented?

- the same type as the parameter
- N-tuples of the same type as the parameter
- a field of the same type as the parameter

5.2.8

In-out parameters

By default, the function parameters are constants. Attempting to change the value of a function parameter in the body of the function will cause a compilation error. If you want the function to modify the value of a parameter, and you want these changes to persist after the function call is completed, define this parameter as in-out. To do this, add the keyword inout before the parameter type. As an in-out parameter, you can only pass a variable to the function, not a constant. In-out parameters cannot have default values and varied parameters cannot be marked as inout:

```
func swapTwoInts (_ a: inout Int, _ b: inout Int) {
  let tempA = a
  a = b
  b = tempA
}
```

The names of someInt and anotherInt must be prefixed with an ampersand when passed to the swapTwoInts (_: _ :) function:

```
var someInt = 3
var anotherInt = 107
swapTwoInts (&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now
\(anotherInt)")
// prints "someInt is now 107 and anotherInt is now 3"
```

5.2.9

What character do we write before the inout variable of the function parameter when calling it?

- * (asterisk)
- # (hash)
- & (ampersand)

5.3 Types of functions

5.3.1

Types of functions

Each function has its specific type, composed of parameter types and the return type of the function:

```
func addTwoInts (_ a: Int, _ b: Int) -> Int {
  return a + b
}
func multiplyTwoInts (_ a: Int, _ b: Int) -> Int {
  return a * b
}
```

This example defines two simple math functions called addTwoInts and multiplyTwoInts. Each of these functions takes two Int values and returns the Int value that results from performing the corresponding mathematical operation.

The type of both of these functions is (Int, Int) -> Int.

Here is another example of a function without parameters and return values:

```
func printHelloWorld () {
  print("hello, world")
}
```

The type of this function is () -> Void.

5.3.2

How is a function type defined?

- only types of all parameters
- return type only
- types of all parameters and the type of return value

5.3.3

Using feature types

Use function types just like any other type in Swift. For example, you can define a constant or variable to be a function type and assign that function to that variable:

```
var mathFunction: (Int, Int) -> Int = addTwoInts
```

The addTwoInts (_: _:) function has the same type as the mathFunction variable, so this assignment is enabled by a Swift type check.

You can now call the assigned function named mathFunction:

```
print ("Result: \(mathFunction(2, 3))")
// Prints "Result: 5"
```

"Another function with the same type can be assigned to the same variable:

```
mathFunction = multiplyTwoInts
```

```
print ("Result: \(mathFunction(2, 3))")
// Prints "Result: 6"
```

As with any other type, you can let Swift derive the function type when assigning a function to a constant or variable:

```
let anotherMathFunction = addTwoInts
// anotherMathFunction is of derived type (Int, Int) -> Int
```

🕮 5.3.4

Types of functions as types of parameters

You can use the function type eg (Int, Int) -> Int as the parameter type for another function. This allows you to keep some aspects of the function implementation for the function caller who provides the callback:

```
func printMathResult(_ mathFunction: (Int, Int) -> Int, _ a:
Int, _ b: Int) {
  print ("Result: \(mathFunction(a, b))")
}
printMathResult (addTwoInts, 3, 5)
// Prints "Result: 8"
```

When printMathResult (_: _: _:) is called, the addTwoInts (_: _:) function is passed and the integer values 3 and 5. The function calls the provided function with values 3 and 5 and prints the result 8.

🕮 5.3.5

Types of functions as return types

You can use a function type as the return type of another function. To do this, type the full type of function immediately after the return arrow (->) of the return function:

```
func stepForward(_ input: Int) -> Int {
  return input + 1
}
func stepBackward(_ input: Int) -> Int {
  return input - 1
}
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
  return backward ? stepBackward : stepForward
}
var currentValue = 3
let moveNearerToZero = chooseStepFunction (backward:
currentValue> 0)
// moveNearerToZero now references the stepBackward() function
```

This function can now be used to count to zero:

```
print("Counting to zero:")
// Counting to zero:
while currentValue != 0 {
  print ("\(currentValue) ...")
  currentValue = moveNearerToZero(currentValue)
}
print ("zero!")
// 3 ...
// 2 ...
// 1 ...
// zero!
```

5.3.6

Can a function in Swift return another function as its return value?

- Yes
- No

5.3.7

Nested functions

All the functions you have encountered in this chapter so far have been examples of global functions that are defined on a global scale. You can also define functions within the bodies of other functions - nested functions.

Nested functions are hidden from the outside world by default, but they can still be called and used in the function where they are defined. This function can also return one of its nested functions so that the nested function can be used outside the function where the nested functions are defined:

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
  func stepForward(input: Int) -> Int {return input + 1}
  func stepBackward(input: Int) -> Int {return input - 1}
  return backward ? stepBackward : stepForward
}
var currentValue = -4
let moveNearerToZero = chooseStepFunction(backward:
currentValue> 0)
// moveNearerToZero now references the stepForward() nested
function
while currentValue != 0 {
  print ("\(currentValue) ...")
  currentValue = moveNearerToZero(currentValue)
}
print ("zero!")
// -4 ...
// -3 ...
// -2 ...
// -1 ...
// zero! "
```

5.3.8

Is the following code correct?

```
func test (a: Int) -> Int {
  func subtest(a: Int) -> Int {a * 2}
  return subtest(a: a) * 3
}
print(subtest(a: 2))
```

- No
- Yes

Closure

Chapter 6

6.1 Closures

🕮 6.1.1

Closures

Closures are separate blocks of code that can be passed and used in your code. The closures in Swift are similar to the blocks in C and Objective-C. Closures can capture and store references to any constants and variables from the context in which they are defined.

Closures have one of three forms:

Global functions are closures that have a name and do not capture any values.

Nested functions are closures that have a name and can capture values from their closure function.

Closure expressions are unnamed closures written in light syntax that can capture values from the surrounding context.

🕮 6.1.2

Sorting method

The standard Swift library provides a method called sorted(by:), which sorts a series of values of a known type based on the output of the sort closures that you specify. Once the sorting process is complete, the sorted(by:) method returns a new field of the same type and size as the old one, with the elements in the correct sorted order.

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
```

The sorted(by:) method accepts a closure that takes two arguments of the same type as the contents of the array and returns a Bool value that indicates whether the first value should appear before or after the second value. The sort closure must return true if the first value should appear before the second value, otherwise false.

In this example, the sort field is a String value, so the sort closure must be a function of type (String, String) -> Bool.

One way to ensure that the sort is closed is to write a normal function of the correct type and pass it as an argument to the sorted method:

```
func backward(_ s1: String, _ s2: String) -> Bool {
  return s1 > s2
}
var reversedNames = names.sorted(by: backward)
// reversedNames equals ["Ewa", "Daniella", "Chris", "Barry",
"Alex"]
```

6.1.3

Syntax of the closure expression

The syntax of the closing expression has the following general form:

```
{(parameters) -> return_type in
  body closures
}
```

Parameters in the closure expression syntax can be input and output parameters, but they cannot have a default value. Variadic parameters can be used if you name them. N-tuples can also be used as parameter types and return types.

```
reversedNames = names.sorted(by: {(s1: String, s2: String) ->
Bool in
  return s1 > s2
})
```

Because the body of the closure is so short, it can be written on one line:

```
reversedNames = names.sorted (by: {(s1: String, s2: String) ->
Bool in return s1> s2})
```

6.1.4

Can closure parameters have default values?

- no
- yes

6.1.5

Derivation of a type from a context

Because the sort closure is passed as an argument to the method, Swift can derive the types of its parameters and the type of value it returns. The sorted (by:) method is called on an array of strings, so its argument must be a function of type (String, String) -> Bool. This means that the types (String, String) and Bool do not need to be written as part of the expression definition. Because all types can be derived, you can also omit the return arrow (->) and parentheses around parameter names:

```
reversedNames = names.sorted (by: {s1, s2 in return s1 > s2})
```

6.1.6

Default return value

Closures with a single expression in the body can return a result without the return keyword by default:

```
reversedNames = names.sorted(by: {s1, s2 in s1 > s2})
```

6.1.7

Automatic argument names

Swift automatically provides argument names for embedded closures that can be used to reference argument values named $0, $1, $2, etc. If you use these shortcut names in your closure, you can omit the argument list from the definition and the number and type of shortcut names will be derived from the expected type of function. The keyword in can also be omitted because the closing expression consists exclusively of its body:

```
reversedNames = names.sorted(by: {$0 > $1})
```

6.1.8

How are automatic closing arguments named?

- #0, #1, #2
- #1, #2, #3
- \$0, \$1, \$2
- \$1, \$2, \$3

🕮 6.1.9

Operator methods

In fact, there is an even shorter way to write the expression before. Swift type String defines its implementation of the greater than (>) operator for a particular string as a method that has two parameters of type String and returns a value of type Bool. This corresponds exactly to the type of method required by the sorted (by:) method. Therefore, you can simply pass an operator greater than and Swift infers that you want to use its string-specific implementation:

```
reversedNames = names.sorted(by: >)
```

6.2 Types of closures

🕮 6.2.1

Trailing Closures

If you need to pass a closure expression to a function as the function’s final argument and the closure expression is long, it can be useful to write it as a *trailing closure* instead. You write a trailing closure after the function call’s parentheses, even though the trailing closure is still an argument to the function. When you use the trailing closure syntax, you don’t write the argument label for the first closure as part of the function call. A function call can include multiple trailing closures; however, the first few examples below use a single trailing closure.

```
func someFunctionThatTakesAClosure (closure :() -> Void) {
  // the body of the function goes here
}
// Here's how to call this function without using endpoint:
someFunctionThatTakesAClosure (closure: {
  // the closure body goes here
})
// Here's how to call this function with an end cap instead:
someFunctionThatTakesAClosure () {
  // the trailing closure's body goes here
}
```

The string sort can be written as a trailing closure

```
reversedNames = names.sorted() {$0 > $1}
```

If the closure expression is provided as a single argument to a function or method, and you specify that expression as an end closure, you do not need to write a pair of parentheses() after the function or method name when calling the function:

```
reversedNames = names.sorted {$0 > $1}
```

📖 6.2.2

Multiple trailing closures

If a function takes multiple closures, you omit the argument label for the first trailing closure and you label the remaining trailing closures. For example, the function below loads a picture for a photo gallery:

```
func loadPicture (from server: Server, completion: (Picture) -
> Void, onFailure: () -> Void) {
  if let picture = download ("photo.jpg", from: server) {
    completion(picture)
  } else {
    onFailure()
  }
}
```

The first closure is a completion handler that displays the image after a successful download. The second closure is the error handler, which displays the error to the user.

```
loadPicture(from: someServer) {picture in
  someView.currentPicture = picture
} onFailure: {
  print("The following image cannot be downloaded.")
}
```

📖 6.2.3

Capturing Values

A closure can *capture* constants and variables from the surrounding context in which it’s defined. The closure can then refer to and modify the values of those constants and variables from within its body, even if the original scope that defined the constants and variables no longer exists.

In Swift, the simplest form of a closure that can capture values is a nested function, written within the body of another function. A nested function can capture any of its outer function’s arguments and can also capture any constants and variables defined within the outer function.

```
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementer () -> Int {
        runningTotal += amount
        // the incrementer function accesses the runningTotal variable and the amount parameter of the function that defines it
        return runningTotal
    }
    return incrementer
}
```

6.2.4

Can a nested function change the variables that are defined in the function that defines the nested function?

- No
- Yes

6.2.5

Closures are reference types

We can write:

```
let incrementByTen = makeIncrementer(forIncrement: 10)
let incrementBySeven = makeIncrementer(forIncrement: 7)
```

incrementBySeven and incrementByTen are constants, but the closures referenced by these constants are still able to increment the runningTotal variables they captured. This is because functions and closures are reference types.

```
let alsoIncrementByTen = incrementByTen
alsoIncrementByTen()
// returns the value 50
incrementByTen()
// returns the value 60
```

The example above shows that calling alsoIncrementByTen is the same as calling incrementByTen. Because they both refer to the same closure, they both increment and return the same sum.

6.2.6

Escaping closures

A closure is said to *escape* a function when the closure is passed as an argument to the function, but is called after the function returns. When you declare a function that takes a closure as one of its parameters, you can write `@escaping` before the parameter's type to indicate that the closure is allowed to escape.

One way that a closure can escape is by being stored in a variable that's defined outside the function. As an example, many functions that start an asynchronous operation take a closure argument as a completion handler. The function returns after it starts the operation, but the closure isn't called until the operation is completed—the closure needs to escape, to be called later. For example:

```
var completionHandlers = [() -> Void] ()
func someFunctionWithEscapingClosure (completionHandler:
@escaping () -> Void) {
  completionHandlers.append (completionHandler)
}
```

The `someFunctionWithEscapingClosure(_:)` function takes a closure as its argument and adds it to an array that's declared outside the function. If you didn't mark the parameter of this function with `@escaping`, you would get a compile-time error.

6.2.7

How do you mark a closure that is allowed to escape?

- @esc
- @escape
- @escaping

📖 6.2.8

Autoclosure

An autoclosure lets you delay evaluation, because the code inside isn't run until you call the closure. Delaying evaluation is useful for code that has side effects or is computationally expensive, because it lets you control when that code is evaluated. The code below shows how a closure delays evaluation:

```
var customersInLine = ["Chris", "Alex", "Ewa", "Barry",
"Daniella"]
print(customersInLine.count)
// prints "5"

let customerProvider = {customersInLine.remove (at: 0)}
print (customersInLine.count)
// prints "5"

print ("Now \(customerProvider()) is serving!")
// prints "Now Chris is serving!"
print(customersInLine.count)
// prints "4"
```

You get the same behavior with delayed evaluation when you pass a closing as an argument to a function:

```
// customersInLine contains ["Alex", "Ewa", "Barry",
"Daniella"]
func serve (customer customerProvider: @autoclosure () ->
String) {
  print ("Now \(customerProvider ()) is serving!")
}
serve(customer: customersInLine.remove(at: 0))
// prints "Now Alex is serving!"
```

📝 6.2.9

How do you mark an autoclosure?

- @auto
- @autoclose
- @autoclosure

6.2.10

Autoclosure

If you want an autoclosure that is allowed to escape, use the @autoclosure and @escaping attributes:

```
// customersInLine contains ["Ewa", "Barry", "Daniella"]
var customerProviders: [() -> String] = []
func collectCustomerProviders (_ customerProvider:
@autoclosure @escaping () -> String) {
  customerProviders.append(customerProvider)
}
collectCustomerProviders(customersInLine.remove (at: 0))
collectCustomerProviders(customersInLine.remove (at: 0))
print("" Collected \(customerProviders.count) closures.")
// prints "Collected 2 closures."
for customerProvider in customerProviders {
  print ("Now \(customerProvider()) is serving!")
}
// Prints "Now Ewa is serving!"
// Prints "Now Barry is serving!"
```

6.2.11

Is it allowed to mark one closure as autoclosure and escaping at the same time?

- yes
- no

Enumerations

Chapter 7

7.1 Enumerations

📖 7.1.1

Enumerations

An *enumeration* defines a common type for a group of related values and enables you to work with those values in a type-safe way within your code.

If you are familiar with C, you will know that C enumerations assign related names to a set of integer values. Enumerations in Swift are much more flexible, and don't have to provide a value for each case of the enumeration. If a value (known as a *raw* value) is provided for each enumeration case, the value can be a string, a character, or a value of any integer or floating-point type.

📖 7.1.2

Enumeration syntax

You introduce enumerations with the `enum` keyword and place their entire definition within a pair of braces:

```
enum CompassPoint {
    case north
    case south
    case east
    case west
}
```

Swift enumeration cases don't have an integer value set by default, unlike languages like C and Objective-C. In the `CompassPoint` example above, `north`, `south`, `east` and `west` don't implicitly equal `0`, `1`, `2` and `3`. Instead, the different enumeration cases are values in their own right, with an explicitly defined type of `CompassPoint`.

Multiple cases can appear on a single line, separated by commas:

```
enum Planet {
    case mercury, venus, earth, mars, jupiter, saturn, uranus,
neptune
}
```

7.1.3

How do you define a new enumeration?

- enum Enum {case nahoru, dolu, doleva, doprava}
- type Enum {case nahoru > dolu > doleva > doprava}
- enum Enum {case nahoru > dolu > doleva > doprava}

7.1.4

Enumeration is a type

Each enumeration definition defines a new type. Like other types in Swift, their names (such as CompassPoint and Planet) begin with a capital letter. Instead, use multiline enumerations to make your code more readable.

```
var directionToHead = CompassPoint.west
```

The directionToHead type is derived at initialization with one of the possible CompassPoint values. Once directionToHead is declared as CompassPoint, you can set it to a different CompassPoint value using shorter syntax:

```
directionToHead = .east
```

7.1.5

Enumeration in the switch statement

You can compare the individual enumeration values in the switch statement:

```
directionToHead = .south
switch directionToHead {
case .north:
  print("Lots of planets have north")
case .south:
  print("Beware of penguins")
case .east:
  print("Where the Sun Rises")
case .west:
  print("Where the sky is blue")
}
// prints "Beware of penguins"
```

7.1.6

Enumeration iteration

For some enumerations, it is useful to have a collection of all their values. To enable this, type: CaseIterable after the enumeration name. Swift makes a collection of all values available as an allCases property of the enumeration type:

```
enum Beverage: CaseIterable {
  case coffee, tea, juice
}
let numberOfChoices = Beverage.allCases.count
print("\(numberOfChoices) drinks available")
// prints "3 drinks available"
for beverage in Beverage.allCases {
  print(beverage)
}
// coffee
// tea
// juice
```

7.1.7

Is the following code valid?

```
enum Enum {case up, down, left, right}
for item in Enum.allCases {
  ...
}
```

- No
- Yes

7.2 Related values, Raw values

7.2.1

Related values

Sometimes it is useful to be able to store values of other types in addition to enumeration values. This additional information is called the related value and differs each time you use this case as a value in your code.

The UPC bar code and QR code will serve as an example. For UPC barcodes we will store n-tuples of four integers and for QR code a string of any length.

In Swift, the list for barcodes for both types of products can look like this:

```
enum barcode {
  case upc (Int, Int, Int, Int)
  case qrCode (String)
}
```

Define an enumeration type called Barcode, which can take either upc values with a related type value (Int, Int, Int, Int), or qrCode values with a related String value.

You can then create new barcodes using one of the following types:

```
var productBarcode = Barcode.upc(8, 85909, 51226, 3)
productBarcode = .qrCode("ABCDEFGHIJKLMNOP")
switch productBarcode {
case .upc (let numberSystem, let manufacturer, let product,
let check):
  print("UPC: \(numberSystem), \(manufacturer), \(product),
\(check).")
case .qrCode(let productCode):
  print("QR code: \(productCode).")
}
// prints "QR code: ABCDEFGHIJKLMNOP."
```

🕮 7.2.2

Simplification of syntax

If all of the associated values for an enumeration case are extracted as constants, or if all are extracted as variables, you can place a single **`var`** or **`let`** annotation before the case name, for brevity:

```
switch productBarcode {
case let .upc (numberSystem, manufacturer, product, check):
  print ("UPC: \(numberSystem), \(manufacturer), \(product),
\(check).")
case let .qrCode (productCode):
  print("QR code: \(productCode).")
}
// prints "QR code: ABCDEFGHIJKLMNOP."
```

7.2.3

Raw values

As an alternative to related values, the enumeration may include pre-populated default values (called raw values) that are of the same type:

```
enum ASCIIControlCharacter: Character {
  case tab = "\t"
  case lineFeed = "\n"
  case carriageReturn = "\r"
}
```

7.2.4

What are the raw values of the following enumeration?

```
enum Enum {case up, down, left, right}
```

- none
- 0, 1, 2, 3
- 1, 2, 3, 4
- case up, down, left, right

7.2.5

Implicitly assigned raw values

When working with enumerations that store integer or string raw values, you do not have to explicitly assign a raw value for each case. If you do not do this, Swift will automatically assign values.

For example, when integers are used for raw values, the default value for each case is one more than the previous case. If the first case does not have a set value, its value is 0:

```
enum Planet: Int {
  case mercury = 1, venus, earth, mars, jupiter, saturn,
uranus, neptune
}
```

In the example above, Planet.mercury has an explicit raw value of 1, Planet.venus has an default raw value of 2, and so on.

7.2.6

What are the raw values of the following enumeration?

```
enum Enum {case up = 2, down, left, right}
```

- 0, 1, 2, 3
- 1, 2, 3, 4
- 2, 3, 4, 5

7.2.7

rawValue properties

When strings are used for raw values, the default value for each case is the text of the case name.

The raw value of the enumeration gets its rawValue property:

```
let earthsOrder = Planet.earth.rawValue
// earthsOrder is 3
let sunsetDirection = CompassPoint.west.rawValue
// sunsetDirection is "west"
```

7.2.8

Initialization from raw value

If you define an enumeration with a raw value type, the enumeration is automatically accepted by the initializer, which takes a raw value type value (as a parameter named rawValue) and returns either the enumeration case or nil. You can use this initializer to try to create a new enumeration instance:

```
let possiblePlanet = Planet(rawValue: 7)
// possiblePlanet is of type Planet? and equals Planet.uranus
```

However, not all possible Int values will find a corresponding planet. For this reason, the raw value initializer always returns an optional enumeration case.

If you try to find a planet with a raw value of 11, the optional Planet value returned by the raw value initializer will be nil:

```
let positionToFind = 11
if let somePlanet = Planet(rawValue: positionToFind) {
  switch somePlanet {
  case .earth:
    print("Suitable for life")
  default:
    print("It's not a safe place for humans")
  }
} else {
  print("There is no planet at position \(positionToFind)")
}
// prints "There is no planet at position 11"
```

7.2.9

What will be the value of the item constant?

```
enum Enum {case up = 2, down, left, right}
let item = Extract(rawValue: 4)
```

- up
- down
- left
- right

7.2.10

Recursive enumerations

A **recursive enumeration** is an enumeration that has another instance of the enumeration as the associated value for one or more of the enumeration cases. You indicate that an enumeration case is recursive by writing `indirect` before it, which tells the compiler to insert the necessary layer of indirection.

```
enum ArithmeticExpression {
  case number(Int)
  indirect case addition(ArithmeticExpression,
ArithmeticExpression)
  indirect case multiplication(ArithmeticExpression,
ArithmeticExpression)
}
```

You can also write indirect before the start of the enumeration. Recursion will apply to all values:

```
indirect enum ArithmeticExpression {
  case number(Int)
  case addition(ArithmeticExpression, ArithmeticExpression)
  case multiplication(ArithmeticExpression,
ArithmeticExpression)
}
```

This enumeration can store three types of arithmetic expressions: simple number, addition, and multiplication of two expressions. Addition and multiplication cases have associated values that are also arithmetic expressions - these associated values allow the nesting of expressions:

```
let five = ArithmeticExpression.number(5)
let four = ArithmeticExpression.number(4)
let sum = ArithmeticExpression.addition(five, four)
let product = ArithmeticExpression.multiplication(sum,
ArithmeticExpression.number(2))
```

7.2.11

Processing of recursive enumerators

A recursive function is a direct way to work with data that has a recursive structure. For example, here is a function that evaluates an arithmetic expression:

```
func evaluate (_ expression: ArithmeticExpression) -> Int {
  switch expression {
  case let .number(value):
    return value
  case let .addition(left, right):
    return evaluate(left) + evaluate(right)
  case let .multiplication(left, right):
    return evaluate(left) * evaluate(right)
  }
}
print(evaluate(product))
// prints "18"
```

This function evaluates a simple number simply by returning an associated value. Evaluates addition or multiplication by evaluating the expression on the left, evaluating the expression on the right, and adding or multiplying them.

Structures and Classes

Chapter 8

8.1 Structures and classes

📖 8.1.1

Structures and classes

Structures and classes are universal, flexible constructions that become the building blocks of your code. You define properties and methods by adding functions to your structures and classes using the same syntax as when defining constants, variables, and functions.

Unlike other programming languages, Swift does not require the creation of custom interfaces and implementation files for custom structures and classes. In Swift, you define a structure or class in one file, and the external interface to that class or structure is automatically made available for use by other code.

📖 8.1.2

Comparison of structures and classes

Structures and classes in Swift have many things in common. Both can:

- Define properties to store values
- Define methods to provide functionality
- Define subscripts to provide access to their values using subscript syntax
- Define initializers to set up their initial state
- Be extended to expand their functionality beyond a default implementation
- Conform to protocols to provide standard functionality of a certain kind

Classes have additional capabilities that structures don’t have:

- Inheritance enables one class to inherit the characteristics of another.
- Type casting enables you to check and interpret the type of a class instance at runtime.
- Deinitializers enable an instance of a class to free up any resources it has assigned.
- Reference counting allows more than one reference to a class instance.

The additional capabilities that classes support come at the cost of increased complexity. As a general guideline, prefer structures because they’re easier to reason about, and use classes when they’re appropriate or necessary. In practice, this means most of the custom data types you define will be structures and enumerations.

8.1.3

Is it recommended to use more structures or use more classes in Swift?

- structures
- classes

8.1.4

Syntax of structures and classes

Structures and classes have a similar definition syntax. You introduce structures with the **`struct`** keyword and classes with the **`class`** keyword. Both place their entire definition within a pair of braces:

```
struct SomeStructure {
    // define structure here
}
class SomeClass {
    // definice class here
}
```

Whenever you define a new structure or class, you define a new Swift type. Give types **`UpperCamelCase`** names (such as **`SomeStructure`** and **`SomeClass`** here) to match the capitalization of standard Swift types (such as **`String`**, **`Int`**, and **`Bool`**). Give properties and methods **`lowerCamelCase`** names (such as **`frameRate`** and **`incrementCount`**) to differentiate them from type names.

```
struct Resolution {
    var width = 0
    var height = 0
}
class VideoMode {
    var resolution = Resolution()
    var interlaced = false
    var frameRate = 0.0
    var name: String?
}
```

The example above defines a new structure called **`Resolution`**, to describe a pixel-based display resolution. This structure has two stored properties called **`width`** and **`height`**. Stored properties are constants or variables that are bundled up and stored as part of the structure or class. These two properties are inferred to be of type **`Int`** by setting them to an initial integer value of **`0`**.

The example above also defines a new class called `VideoMode`, to describe a specific video mode for video display. This class has four variable stored properties. The first, `resolution`, is initialized with a new `Resolution` structure instance, which infers a property type of `Resolution`. For the other three properties, new `VideoMode` instances will be initialized with an `interlaced` setting of `false` (meaning "noninterlaced video"), a playback frame rate of `0.0`, and an optional `String` value called `name`. The `name` property is automatically given a default value of `nil`, or "no `name` value", because it's of an optional type.

8.1.5

Which keywords are used to define structures and classes?

- structure and class
- struct and classes
- struct and class

8.1.6

Structure and class instances

The Resolution definition and the VideoMode class definition only describe what the Resolution or VideoMode will look like. They do not describe the specific resolution or video mode themselves. To do this, you must instantiate a structure or class:

```
let someResolution = Resolution()
let someVideoMode = VideoMode()
```

Structures and classes use initializer syntax for new instances. The simplest form of initializer syntax uses a class or structure type name followed by empty parentheses, such as Resolution() or VideoMode(). This creates a new instance of the class or structure with any properties initialized to their default values.

8.1.7

How do you create a new instance of a structure called MyStructure with its default properties?

- new MyStructure
- MyStructure()

- MyStructure[]

8.1.8

Accessing Properties

You can access the properties of an instance using *dot syntax*. In dot syntax, you write the property name immediately after the instance name, separated by a period (`.`), without any spaces:

```
print("The width of someResolution is
\(someResolution.width)")
// Print "The width of someResolution is 0"
```

In this example, **`someResolution.width`** refers to the **`width`** property of **`someResolution`**, and returns its default initial value of `0`.

You can drill down into subproperties, such as the **`width`** property in the **`resolution`** property of a **`VideoMode`**:

```
print("The width of someVideoMode is
\(someVideoMode.resolution.width)")
// prints "The width of someVideoMode is 0"
```

You can also use dot syntax to assign a new value to a variable property:

```
SomeVideoMode.resolution.width = 1280
print("The width of someVideoMode is now
\(someVideoMode.resolution.width)")
// prints "The width of someVideoMode is now 1280"
```

8.1.9

How do you access the width property of myInstance instance?

- myInstance.width
- myInstance -> width
- width.myInstance

8.1.10

Initializers for structure types with values

All structures have an automatically generated initializer that you can use to initialize the properties of new instances of the structure:

```
let vga = Resolution(width: 640, height: 480)
```

8.1.11

Is a class initializer automatically available that sets class properties?

- Yes
- No

8.2 Types of values, Indicators

8.2.1

Structures and Enumerations Are Value Types

A *value type* is a type whose value is *copied* when it's assigned to a variable or constant, or when it's passed to a function.

You've actually been using value types extensively throughout the previous chapters. In fact, all of the basic types in Swift—integers, floating-point numbers, Booleans, strings, arrays and dictionaries—are value types, and are implemented as structures behind the scenes.

Consider this example, which uses the Resolution structure from the previous example:

```
let hd = Resolution(width: 1920, height: 1080)
var cinema = hd
```

This example declares a constant called `hd` and sets it to a `Resolution` instance initialized with the width and height of full HD video (1920 pixels wide by 1080 pixels high).

It then declares a variable called `cinema` and sets it to the current value of `hd`. Because `Resolution` is a structure, a *copy* of the existing instance is made, and this new copy is assigned to `cinema`. Even though `hd` and `cinema` now have the same width and height, they're two completely different instances behind the scenes.

```
cinema.width = 2048
print("cinema is now \(cinema.width) pixels wide")
// prints "cinema is now 2048 pixels wide"
```

```
print ("hd is still \(hd.width) pixels wide")
// prints "hd is still 1920 pixels wide"
```

When **`cinema`** was given the current value of **`hd`**, the *values* stored in **`hd`** were copied into the new **`cinema`** instance. The end result was two completely separate instances that contained the same numeric values. However, because they're separate instances, setting the width of **`cinema`** to **`2048`** doesn't affect the width stored in **`hd`**

8.2.2

Does assigning a structure instance to another variable create a copy of the instance?

- yes
- no

8.2.3

Value types - enumeration assignment

The same behavior applies to enumerations:

```
enum CompassPoint {
    case north, south, east, west
    mutating func turnNorth() {
        self = .north
    }
}
var currentDirection = CompassPoint.west
let rememberedDirection = currentDirection
currentDirection.turnNorth()
print ("Current direction is \(currentDirection)")
print ("Remembered direction is \(rememberedDirection)")
// prints "Current direction is north"
// prints "Remembered direction is west"
```

When the rememberedDirection parameter is assigned the currentDirection value, it is actually set to a copy of that value. Subsequent changes to the currentDirection value do not affect the copy of the original value that was stored in rememberedDirection.

8.2.4

Can an enumeration contain methods (functions)?

- Yes
- No

8.2.5

Classes Are Reference Types

Unlike value types, *reference types* are *not* copied when they're assigned to a variable or constant, or when they're passed to a function. Rather than a copy, a reference to the same existing instance is used.

Here's an example, using the **`VideoMode`**class defined earlier:

```
let tenEighty = VideoMode ()
tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
tenEighty.frameRate = 25.0
```

This example declares a new constant called **`tenEighty`** and sets it to refer to a new instance of the **`VideoMode`** class. The video mode is assigned a copy of the HD resolution of `1920` by `1080` from before. It's set to be interlaced, its name is set to `"1080i"`, and its frame rate is set to `25.0` frames per second.

Next, **`tenEighty`** is assigned to a new constant, called **`alsoTenEighty`**, and the frame rate of **`alsoTenEighty`** is modified:

```
let alsoTenEighty = tenEighty
alsoTenEighty.frameRate = 30.0
```

Because classes are reference types, **`tenEighty`** and **`alsoTenEighty`** actually both refer to the *same* **`VideoMode`** instance. Effectively, they're just two different names for the same single instance

```
print("The frameRate tenEighty property is now
\(tenEighty.frameRate)")
// prints "The frameRate tenEighty property is now 30.0"
```

Note that **`tenEighty`** and **`alsoTenEighty`** are declared as *constants*, rather than variables. However, you can still change **`tenEighty.frameRate`** and **`alsoTenEighty.frameRate`** because the values of the `tenEighty` and `alsoTenEighty` constants themselves don't actually change. `tenEighty` and `alsoTenEighty` themselves don't "store" the `VideoMode` instance—instead, they both *refer* to a `VideoMode` instance behind the scenes. It's the `frameRate` property of the underlying `VideoMode` that's changed, not the values of the constant references to that `VideoMode`.

8.2.6

Does assigning a class instance to another variable create a copy of the instance?

- Yes
- No

8.2.7

Identity operators

Because classes are reference types, it's possible for multiple constants and variables to refer to the same single instance of a class behind the scenes. (The same isn't true for structures and enumerations, because they're always copied when they're assigned to a constant or variable or passed to a function.)

It can sometimes be useful to find out whether two constants or variables refer to exactly the same instance of a class. To enable this, Swift provides two identity operators:

Identical to (===)
Not identical to (!==)

Use these operators to check whether two constants or variables refer to the same single instance:

```
if tenEighty === alsoTenEighty {
    print("tenEighty and alsoTenEighty refer to the same
VideoMode instance.")
}
// prints "tenEighty and alsoTenEighty refer to the same
VideoMode instance."
```

8.2.8

What operator is used to compare the same instance of a class?

- =
- ==
- ===

8.2.9

Pointers

If you have experience with C, C++, or Objective-C, you may know that these languages use *pointers* to refer to addresses in memory. A Swift constant or variable that refers to an instance of some reference type is similar to a pointer in C, but isn't a direct pointer to an address in memory, and doesn't require you to write an asterisk (*) to indicate that you are creating a reference. Instead, these references are defined like any other constant or variable in Swift. The standard library provides pointer and buffer types that you can use if you need to interact with pointers directly

8.2.10

Do you need to indicate a reference to a class instance (pointer) with an *(asterisk)?

- No
- Yes

PRISCILLA

priscilla.fitped.eu