# Swift Basics

This workbook provides an introduction to the basics of the Swift programming language. It includes explanations, code descriptions and code examples, which you should read. It also includes activities, for example to create or modify some code. The activities reinforce what you have read, but also give you vital coding practice. The workbook is design to be read in sequence, if you skip parts it may make it harder to understand later parts.

Whilst the workbook includes everything you need to cover, you may find it useful to refer to other sources of information. There is a lot of information and documentation available at <https://swift.org/documentation/> including a guide <https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html>

The codeacademy website has a course on Swift, but it is new and doesn’t contain many lessons at the moment. Our library also has books on Swift.

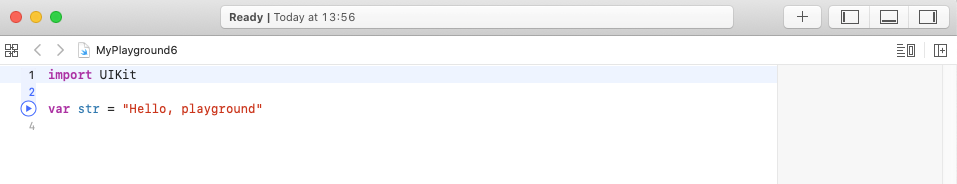
There are different versions of Xcode and Swift, so watch out for incompatibilities!

## Playground

Xcode includes the Playground, an interactive environment where Swift code can be written and executed in real-time. It is a useful space for experimenting with bits of Swift code before using them in an app.

1. Start Xcode, then select File > New > Playground
2. Select Blank
3. Enter the filename and folder name you wish to use, and click “create”

You should see something like this:



The first thing we can see is the UIKit module being imported. Don’t worry about this – it is just importing some standard library functions.

The next line is a variable declaration. The variable str is set to the value “Hello, playground”.

1. Add the following code on the line following the variable declaration (line 4 in the above screenshot):

print(str)

1. Run the code using the play/stop icon towards the bottom of the Playground window. You can show or hide the output using the left-hand icon.



You will notice that the grey panel to the right shows the value the variable is set to and the string which will be printed out by the print statement (note the added \n newline character). We also now have the option to run our code using the play button towards the bottom of the screen.

1. Add the following code on the line following the print statement (line 5 in the above screenshot):

print(str)

1. Run the code – we can see the effect of the newline.

### Comments

Swift supports single line and multiline comments:

// this is a single line comment

/\*

This is a

Multiline comment

\*/

Unlike many programming languages, Swift also allows comments to be nested:

/\*

This is a

Multiline comment

/\*

This is a nested

Multiline comment

\*/

\*/

This is handy for commenting out blocks of code that already contain comments.

We can also add single line comments to the end of lines of code, for example:

Var str = "Hello, playground" // comment here

### Terminators

One thing we might notice at this point is that our statements have no terminator character, such as a semi-colon. In fact, Swift does use the semi-colon as a terminator character, it’s just that in most cases they are not necessary. We could include a semi-colon at the end of any of our statements, or all of them, if we wanted to for example:

Var str = "Hello, playground";

print(str)

print(str);

The only time we have to use one is if we have two or more statements on the same line, for example:

Var str = "Hello, playground"

print(str); print(str)

### Variables

Swift provides several types of basic variable:

|  |  |  |
| --- | --- | --- |
| **keyword** | **description** | **examples** |
| String | a string of characters | "Hello" "3" "3.0" "" |
| Character | a single character | "h" "3" "\*" |
| Int | an integer | 3 300 330 |
| Float | a 32 bit floating point number | 3.45 3.0 |
| Double | a 64 bit floating point number | 3.45 3.0 |
| Bool | a Boolean | true false |

All variables must have a type. Once we have defined the type of a variable, we cannot change that type.

We can declare variable with or without assigning them a value. To declare a variable without assigning a value, we specify its name and type, for example

var myVariable: Bool

Note the use of the colon between the variable name and the variable type.

If we declare a variable by assigning a value (and not declaring a type) then Swift will infer the type of the variable, for example:

var myVariable1 = 3 // infers Int type

var myVariable2 = "hello" // infers String type

There are a couple of inferred types which might trip us up:

var myVariable3 = "3" // infers String type

var myVariable4 = "h" // infers String type

var myVariable5 = 3.0 // infers Double type

If we aren’t sure what the type of a variable is, we can alt+left-click on the variable name and the variable type will appear in a pop-up box.

We can define the type and the value at the same time, for example:

var myVariable5: Float = 3

Variable types are important in Swift, for example, we cannot add an Int and a Float.

1. Define a Float type variable called myFloat and give it the value 3.5
2. Define an Int type variable called myInt and give it the value 3
3. Define a new variable, called mySum like this:

var mySum = myFloat + myInt

and expect to see an error message

If we want to add these two numbers together we will need to cast one of them into the same type as the other. So, we could cast the Int to be a Float, or we could cast the Float to be an Int, these will have different effects and the difference is important.

1. Change the definition of mySum to be this:

var mySum = myFloat + Float(myInt)

This casts (converts) myInt to be a Float and the sum can now be calculated. The total will be 6.5

1. Now change the definition of mySum to be this:

var mySum = Int(myFloat) + myInt

This casts myFloat to be an Int and the sum can now be calculated. The total will be 6, NOT 6.5! When we cast a Float or Double to an Int the fractional parts are discarded.

1. Consider the following code:

var myFloat1: Float = 3.5

var myFloatr2: Float = 3.5

print(myFloat1 + myFloat2) // sum one

print(Int(myFloat1) + Int(myFloat2)) // sum two

print(Int(myFloat1 + myFloat2)) // sum three

1. Predict the value that will be printed by sum one, sum two and sum three.
2. Run the code and see if your predictions were correct. If not, work out why. Note that the print statement has actually cast the resulting numbers as strings as well.
3. Try the following piece of code:

print(String(myFloat1) + String(myFloat2))

Note that adding Strings together concatenates them.

1. We can include the value of variables in a String in another way, by using String interpolation. Try the following piece of code:

print("The value of myFloat1 is \(myFloat1) at the moment")

### Constants

We can define constants in Swift using the let statement. A constant must have a value when it is defined. It may have a type also. If the type is not specified, it will be inferred.

1. Define couple of constants using the following code:

let myConstant1 = 3

let myConstant2 = 3.0

1. Check the types of these constants, using alt+left-click. If we had wanted myConstant1 or myConstant2 to be a Float instead, we would have needed to use:

let myConstant1: Float = 3

let myConstant2: Float = 3.0

1. Once we have defined a constants value, it cannot be changed. Try the following:

myConstant1 = 5

### Flow Control

We can use both Ifs and Switches in Swift

##### If

The if statement in Swift is straightforward, as shown below:

if myAge > 65 {

print("Pensioner")

}

We can include a default else statement:

if myAge > 65 {

print("Pensioner")

} else {

print("Working Age")

}

We can also include nested if statements:

if myAge > 65 {

print("Pensioner")

} else if myAge > 18 {

print("Working Age")

} else {

print("Youth")

}

The standard comparison operators are available:

|  |  |  |
| --- | --- | --- |
| **operator** | **example** | **meaning** |
| == | x == y | x is equal to y |
| != | x != y | x is not equal to y |
| > | x > y | x is greater than y |
| < | x < y | x is less than y |
| >= | x >= y | x is greater than or equal to y |
| <= | x <= y | x is less than or equal to y |

1. Modify the above if statement example to include the following additional age ranges: 5 to 18, School Age; 0 to 4, Infant; 100, Centenarian, less than 0, Error.

##### Switch

As an alternative to multiple if-else statements we can use a Switch. Common types of switch in Swift are switch on number, switch on character and switch on number range.

The following code example shows a switch on number. The valuable of the variable day is used to determine which case applies.

var day = 2

var dayOfWeek: String

switch day {

case 1:

dayOfWeek = "Monday"

case 2:

dayOfWeek = "Tuesday"

case 3:

dayOfWeek = "Wednesday"

case 4:

dayOfWeek = "Thursday"

case 5:

dayOfWeek = "Friday"

case 6:

dayOfWeek = "Saturday"

case 7:

dayOfWeek = "Sunday"

default:

dayOfWeek = "Unknown"

}

print(dayOfWeek)

1. Try the above code with different values for day to check its behaviour.
2. Following the above example, write a switch statement that maps an integer to the corresponding colour of the rainbow, 1 = red, 2 = orange etc.

The following example shows a switch on Character. It also shows how multiple cases can be included in a single case statement.

var grade = Character

grade = "A"

switch grade {

case "A", "B", "C", "D":

print("Passed")

case "F":

print("Failed")

default:

print("Undefined")

}

1. Try the above code with different values for grade to check its behaviour.
2. Modify the above example so that A and B print "Passed higher" and C and D print "Passed lower"

Every case in a switch statement must contain at least one executable statement (comments are not executable statements). If we don't want to take an action in a particular case because we wish to drop through into the next case (note, it doesn't check the next case, it executes its statements) we have to use the fallthrough statement. The following code has exactly the same effect as the previous example.

var grade = Character

grade = "A"

switch grade {

case "A":

fallthrough

case "B":

fallthrough

case "C":

fallthrough

case "D":

print("Passed")

case "F":

print("Failed")

default:

print("Undefined")

}

1. Try the above code with different values for grade to check its behaviour.
2. Modify the above example so that A and B print "Passed higher" and C and D print "Passed lower"

We can include other executable statements in a case before using a fallthrough, as shown below.

Note also in this example, that we are doing something different with the print statement. Recall that the print statement defaults to inserting a "\n" newline at the end of the string. We can change this behaviour using the terminator. If we set the terminator to "" it will print the string with no "\n". If we set it to " " it will print the string followed by a space and no "\n". If we set it to "\n" it will be exactly the same as not specifying a terminator at all.

var grade = Character

grade = "A"

switch grade {

case "A":

print("Excellent! ", terminator="")

fallthrough

case "B":

print("Well done! ", terminator="")

fallthrough

case "C":

print("Good! ", terminator="")

fallthrough

case "D":

print("You have passed")

case "F":

print("Failed")

default:

print("Undefined")

}

1. Try the above code with different values for grade to check its behaviour.
2. Modify the above example so that A prints "Excellent! Well done!", B prints "Well done" and C and D print "Passed lower"

We can also switch on a range of numbers, as show below.

var percentage = 67

switch percentage {

case 0...39:

print("Fail")

case 40...100:

print("Pass")

default:

print("Invalid percentage")

}

1. Try the above code with different values for grade to check its behaviour.
2. Modify the above example so that 40-49 prints "Pass", 50-59 prints "2.2", 60-69 prints "2.1" and 70-100 prints "First",

### Looping

Swift provides several different looping statements.

##### For-in

For-in loops allow us to loop through the items in a defined list of items. The simplest example of this is to loop through a range of numbers, for example:

for i in 0...4 {

print(i, terminator: " ")

}

1. Using for-in loops, print out the times tables for numbers up to 10, e.g.

1 x 1 = 1

1 x 2 = 2

1 x 3 = 3

1 x 4 = 4

1 x 5 = 5

1 x 6 = 6

1 x 7 = 7

1 x 8 = 8

1 x 9 = 9

1 x 10 = 10

==========

2 x 1 = 2

2 x 2 = 4

…

10 x 10 = 100

Hints: Use nested for-in loops. Remember \()

##### While

While loops loop whilst a condition is true. The condition is checked before any statements are executed. For example:

var i = 0

while i < 10 {

print(i)

i = i + 1 // we need to update the counter ourselves

}

1. Change the above code to only print even numbers between 1 and 10.

##### Repeat-while

Repeat-while loops also loop whilst a condition is true. The condition is checked after any statements are executed, so the statements will be executed at least once. For example

var i = 0

repeat {

print(i)

i = i + 1 // we need to update the counter ourselves

} while i < 10

1. Change the above code to only print odd numbers between 10 and 1.

The difference between while and repeat is illustrated in the example below:

var i = 0

while i > 0 {

print("Executed while loop")

}

repeat {

print("Executed repeat loop")

} while i > 0

The condition in both the while and repeat loops is false, because i is not greater than zero (it is equal to zero). However, one of the print statements will be executed – you should be able to work out which one, run the code the check if you were right.

Hint: You can clear the output area of the playground using cmd-k

### Control transfer

Swift also provides two control transfer statements, break and continue. These allow us to break out of a loop, or to continue to the next iteration of a loop. At this point, we will only consider the break statement.

Suppose we have a string of characters, "This is a string", and we want to find the first occurrence of a specific character, "a", within that string, which is 8 (remember we start at 0). We could write the following code. Note that we are looping through the characters in a string and that index+=1 is the same as index = index + 1

var c: Character

var found = false

var index = 0

for c in "This is a string" {

if c != "a" && !found {

index+=1

} else {

found = true

}

}

print("Position of 'a' is \(index)")

However, this code has the disadvantage that we continue looping through the string and checking characters, even after we have found a match. If we use a break statement, we can instead exit the loop as soon as we have found a match:

var c: Character

var index = 0

for c in "This is a string" {

if c == "a" {

break

} else {

index+=1

}

}

print("Position of 'a' is \(index)")

1. Run both of the examples above and make sure you understand how they work.
2. Change each of the examples so that the character and string are both held in variables and so that if the character is not found in the string a message "Character not found in string" is printed.

The break statement can also be useful with switches, consider this example from earlier:

switch grade {

case "A", "B", "C", "D":

print("Passed")

case "F":

print("Failed")

default:

print("Undefined")

}

What if we didn't want to do anything if the grade was undefined – we need to have at least one executable statement (and comments are an executable statement). The answer is to use a break statement, for example:

switch grade {

case "A", "B", "C", "D":

print("Passed")

case "F":

print("Failed")

default:

break

}

### Collections

Swift offers two types of collections, arrays and dictionaries. In an array, the order of items matters. In a dictionary the order does not matter, instead each item is identified by a unique key. Both arrays and dictionaries may only contain items of the same type, for example an array of integers.

#### Arrays

An array is an indexed collection of items. An array may be either mutable (more items can be added to it) or immutable (the items are fixed on creation).

An immutable array is useful if we have some ordered set of constants, such as days of the week or months of the year. An Immutable array is defined like this:

let daysOfWeek = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]

Most often we will be using mutable arrays. A mutable array can be defined with values but no type, like this:

var myCars = ["Ferrari", "Porsche", "Maserati"]

in which case the type is implied.

Alternatively, we can specify just the data type of the items to be held in the array:

var myCars = [String]()

in which case the array is initially empty.

Or we can specify both the type and the items:

var myCars : [String] = ["Ferrari", "Porsche", "Maserati"]

We can also create an array of a specified size and initialise each item, for example:

var score = [Float](repeating : 0.0, count : 5) // produces [0, 0, 0, 0, 0]

We can retrieve items from an array using their index number (remembering that the index starts from zero), for example:

var myCars = ["Ferrari", "Porsche", "Maserati"]

print(myCars[1]) // would print Porsche

We can change existing items in an array in a similar fashion, for example

myCars[1] = "Lamborghini" // array is now ["Ferrari", "Lamborghini", "Maserati"]

We can insert a new item into an array at a specific index point in the following way:

myCars.insert("Bugatti", at: 1) // array is now ["Ferrari", "Bugatti", "Lamborghini", "Maserati"]

If the index value refers to an "empty spot" at the end of the array (in the above case, index 4) then the new item will be added to the end of the array. However if the index refers to "empty spots" further away than this (5 or more in the above case) an error will occur.

We can append new items to the end of an array like this:

myCars.append("Morgan") // array is now ["Ferrari", "Bugatti", "Lamborghini", "Maserati", "Morgan"]

or like this

myCars += ["Morgan"]

We can also append an array to an existing array (so long as the item types are the same), for example:

myCars += ["Lotus", "Aston Martin"] // array is now ["Ferrari", "Bugatti", "Lamborghini", "Maserati", "Morgan", "Lotus", "Aston Martin"]

We can delete individual items from an array basic on their index value, for example

myCars.remove(at : 2) // array is now ["Ferrari", "Bugatti", "Maserati", "Morgan", "Lotus", "Aston Martin"]

if we want to use the removed value we can do:

var soldCar = myCars.remove(at : 2)

in which case soldCar would contain "Lamborghini"

We can also remove the last item

myCars.removeLast() // array is now ["Ferrari", "Bugatti", "Maserati", "Morgan", "Lotus"]

We can empty an array

myCars.removeAll()

We can check the size (number of items) of the array:

var lengthOfArray = myCars.count

and whether it is empty

var arrayIsEmpty = myCars.isEmpty

There are three typical ways we might want to loop over the items in an array.

Loop over all items

for car in myCars {

print(car)

}

Would print

Ferrari

Bugatti

Maserati

Morgan

Lotus

Loop over items in a specific index range

for index in 0…2 {

print(myCar[index])

}

Would print

Ferrari

Bugatti

Maserati

Loop over both the index value and the item value from the array

for (index, value) in myCars.enumerated() {

print("Item number \(index) is \value")

}

Would print

Item number 0 is Ferrari

Item number 1 is Bugatti

Item number 2 is Maserati

Item number 3 is Morgan

Item number 4 is Lotus

We can test two arrays for equality using == e.g.

areArraysEqual = myArray1 == myArray2

|  |  |  |  |
| --- | --- | --- | --- |
| **myArray1** | **myArray2** | **==** | **explanation** |
| [1, 2, 3, 4] | [1, 2, 3] | false | Different lengths |
| [1, 2, 3, 4] | [1, 2, 3, 5] | false | Different items |
| [1, 2, 3, 4] | [4, 3, 2, 1] | false | Items in different order |
| [1, 2, 3, 4] | [1, 2, 3, 4] | true | Same items in same order |

1. Given two arrays, one holding the days of the week and the other holding the colours of the rainbow, write code which produces a new array which merges the two, producing the following output: ["Monday", "Red", "Tuesday", "Orange" … "Friday", "Violet"]
2. Given an array of integers, write code that sorts them into order (e.g. using a bubble sort)

##### Dictionaries

A dictionary is a collection of items of the same type that is identified using a key. Dictionaries similar to arrays, but cannot access by index, only access by key. Because of this, the order of items in a dictionary is not important.

Consider the following dictionary definition:

var platforms: Dictionary<String, String> = [

"Apple" : "iOS",

"Google" : "Android",

"Microsoft" : "Windows Phone"

]

The dictionary platforms contains three items. Each item is a key/value pair. So "Apple" is the key that contains the value "iOS". In this example, both the key and the value are defined to be of type String.

We can either define the type of the key and value as above, or we can let the type be inferred. The following declaration is equivalent to the above declaration:

var platforms: Dictionary = [

"Apple" : "iOS",

"Google" : "Android",

"Microsoft" : "Windows Phone"

]

The key used in a dictionary need not be a String, for example, it could be an Integer:

var ranking = [

1 : "Gold",

2 : "Silver",

3 : "Bronze"

]

Remember the number in this case is a key, not an index value. It does not determine the order of the items in the Dictionary, as order is irrelevant.

Similarly the value does not need to be a String, for example it could be an Integer:

var numbers : Dictionary = ["one" : 1, "two" : 2, "three" : 3]

T

he value can also be an array, for example

var products = [

"Apple" : ["iPhone", "iPad", "iPod"],

"Google" : ["Nexus 4", "Nexus 5"],

"Microsoft" : ["Lumia 1320", "Lumia 1520"]

]

We can also define empty Dictionaries

var months = Dictionary<Int, String>()

We can then populate the Dictory by specifying the key and value, e.g.

months[1] = "January"

As with Arrays, we can define both mutable Dictionaries (using var) and immutable Dictionaries (using let).

We can check the size of a dictionary (the number of key/value items) like this:

var products = [

"Apple" : ["iPhone", "iPad", "iPod"],

"Google" : ["Nexus 4", "Nexus 5"],

"Microsoft" : ["Lumia 1320", "Lumia 1520"]

]

print(products.count) // will be three

We retrieve a value from a Dictionary by specifying its key. We need to be careful when retrieving values from Dictionaries as we can potentially retrieve a nil value if the key has no value, or if the key does not exist in the Dictionary (technically the value is Optional). We should therefore always check for a nil return before using a value retrieved from a dictionary, for example:

var platforms: Dictionary<String, String> = [

"Apple" : "iOS",

"Google" : "Android",

"Microsoft" : "Windows Phone"

]

var p = platforms["Apple"]

if p != nil {

print(p!)

} else {

print("Key not found")

}

Note, when we are printing p there is an additional explanation mark ! This is because the value is optional and we need to unwrap it.

1. Define the Dictionary platforms as shown earlier.
2. Try the above code with and without the unwrapping exclamation mark so that you can see the difference.

We can modify or add an item in the Dictionary in two ways.

var platforms: Dictionary = [

"Apple" : "iOS",

"Google" : "Android",

"Microsoft" : "Windows Phone"

]

platforms["Microsoft"] = "WinPhone"

If the key exists in the Dictionary, then the value is changed to the new value. If the key does not already exist, then a new item is created with the specified key and value. Remember the order of the items does not matter.

Alternatively we can update in the following way:

platforms.updateValue("Winphone", forKey: "Microsoft")

Again, If the key exists in the Dictionary, then the value is changed to the new value. If the key does not already exist, then a new item is created with the specified key and value.

However, this method has has the advantage of returning the old value of the item, or nil if the key was not found. This allows us to determine whether a new key was inserted or not.

Similarly we can remove an item in two ways:

Platforms["Microsoft"] = nil;

Removes the key/value pair if it exists. Note, it doesn't just set the value to nil.

Alternatively we can remove in the following way:

platforms.removeValueForKey("Microsoft")

Again this method has has the advantage of returning the old value of the item, or nil if the key was not found. This allows us to determine whether an item was deleted or not.

If we want to empty a Dictionary we can use

Platforms = [:]

There are several ways in which we can loop over a Dictionary.

var platforms: Dictionary = [

"Apple" : "iOS",

"Google" : "Android",

"Microsoft" : "Windows Phone"

]

We can loop over items using a for-in loop

for platform in platforms {

print(platform)

}

We can loop on the key and value separately:

for (company, platform) in platforms {

print("\(company) uses the \(platform) platform")

}

We can loop just on the keys:

for company in platforms.keys {

print("Company \(company)")

}

We can loop on just the values:

for platform in platforms.platformss {

print("Platform \(platform)")

}

You can also assign the keys or values of a Dictionary directly to an array:

var companies = platforms.keys

var oses = platforms.values

We can test Dictionaries for equality using == which will return true if they contain the same key/value items. Remember the order is irrelevant.

|  |  |  |  |
| --- | --- | --- | --- |
| **myDictionary1** | **myDictionary2** | **==** | **explanation** |
| 1 : "Gold",  2 : "Silver",  3 : "Bronze" | 1 : "Gold",  2 : "Silver" | false | Different lengths |
| 1 : "Gold",  2 : "Silver",  3 : "Bronze" | 1 : "Ivory",  2 : "Silver",  3 : "Bronze" | false | Different items |
| 1 : "Gold",  2 : "Silver",  3 : "Bronze" | 2 : "Gold",  1 : "Silver",  3 : "Bronze" | false | Different items |
| 1 : "Gold",  2 : "Silver",  3 : "Bronze" | 1 : "Gold",  2 : "Silver",  3 : "Bronze" | true | Same items |
| 1 : "Gold",  2 : "Silver",  3 : "Bronze" | 2 : "Silver",  3 : "Bronze",  1 : "Gold" | true | Same items |

1. Given the following Dictionary definition:

var products = [

"Apple" : ["iPhone", "iPad", "iPod"],

"Google" : ["Nexus 4", "Nexus 5"],

"Microsoft" : ["Lumia 1320", "Lumia 1520"]

]

Write code the print out the following (don't worry about the order of the companies):

Microsoft

========

Lumia 1320

Lumia 1520

Google

========

Nexus 4

Nexus 5

Apple

========

iPhone

iPad

iPod

1. Given an array of odd integers 1 to 9 and another array of odd strings "one" to "nine", write code to create a dictionary of the form [1 : "one", 3 : "three" etc.]