GDAŃSK UNIVERSITY OF TECHNOLOGY

Systemy Operacyjne macOS i iOS

Lab 1

Say hello to Swift!

Gdańsk 2025
Tomasz Idzi

Introduction

In this lab you will start your adventure with programming language use in iOS app development - Swift.

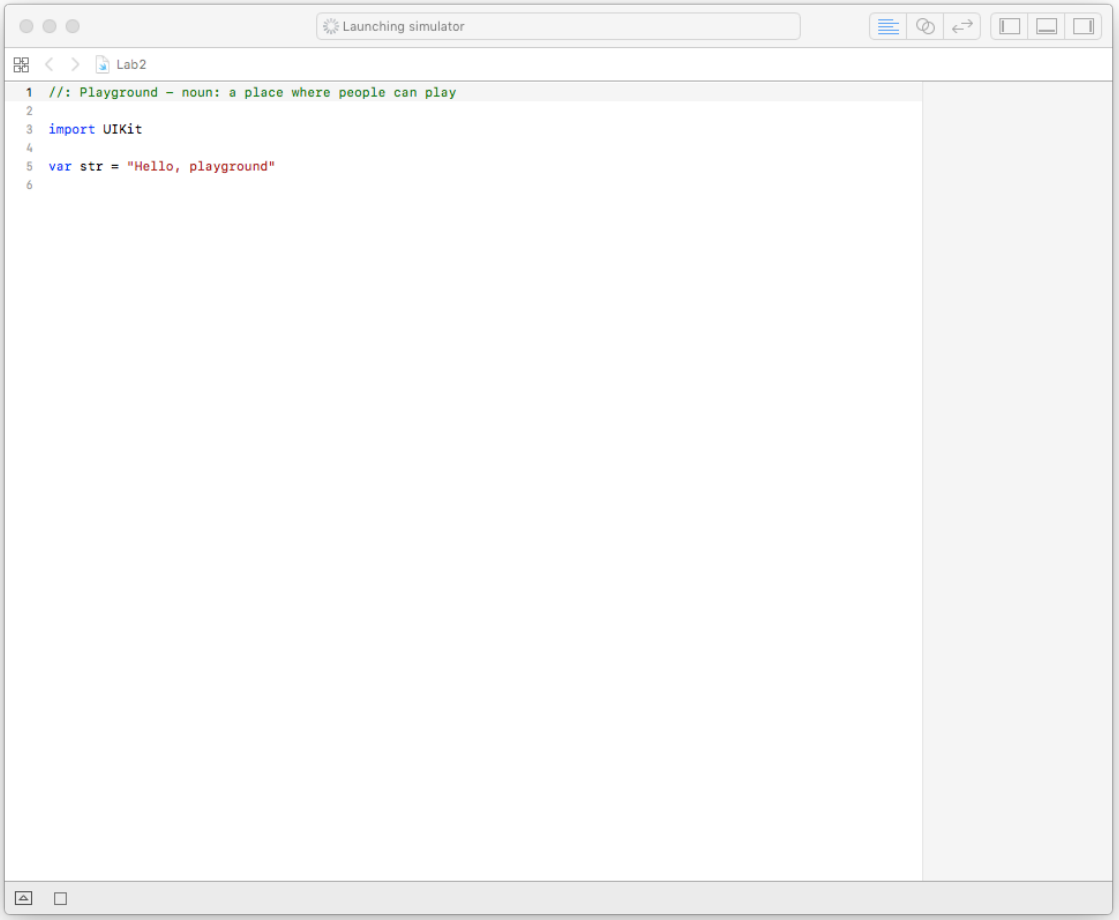
Playground

Before you start making real-life iOS applications, you have to learn the basics of the programming language known as Swift. The best place to start with it will be *Playground*. It's part of the Xcode where you can experiment with programming ideas without building an entire app. You'll write some code, watch it run, think about it a bit, change some lines, and watch it run again. Down the road, you can turn your successful experiments into a fully featured app, but for now, just play—and learn.

WEB VERSION: https://swiftfiddle.com **or**
https://www.programiz.com/swift/online-compiler

Getting Started

Open Xcode app, the select **File**>**New**>**Playground**. Select template - **Blank** and tap **Next**. Choose a place where you would like to save the file and type its name. Then tap **Create**. After that, you should see this window:



Strings and Text (1 point)

A string is an ordered collection of characters, such as “Gdansk University of Technology” or “study, lecture, labs, exam”. In Swift strings are represented by the **String** type. The contents of a **String** can be accessed in various ways, including as a collection of **Character** values.

Before we create a new string, we have to remember that in Swift we have two keyword which define constants - **let** and variables - **var**.

let name = "Name"
var age = 25

It means that we can change the value of ***age,*** but not the value of ***name***.

In Swift it's possible to create a new string by connecting two strings using the **+** operator.

```
let name = "FirstName" + " LastName" // "FirstName LastName"
```

It's also possible to create string based on any combination of values, by using **\(<value>)** syntax.

```
let firstName = "FirstName"
let lastName = "LastName"

let name = "\(firstName) \(lastName)" // "FirstName LastName"
```

You can access the individual **Character** values for a **String** by iterating over it's characters property with for-on loop:

```
var string = "Hello!🙌"

for character in string {
    print(character)
}

// H
// e
// l
// l
// o
// !
// 🙌
```

String and character equality is checked with the "equal to" operator (==) and the "not equal to" operator (!=).

```
let str1 = "Have a nice day! ☀️"
let str2 = "Have a nice day! ☀️"

if str1 == str2 {
    print("Strings are identical.")
}

// Strings are same.
```

Exercise:

1. Create two var, which contain numbers. Compute the sum of this values and print it in following format:
   "5 + 10 = 15"

2. Create a new string “Gdansk University of Technology”, then create another one where all occurrences of character “n” will be replaced with “⭐”. Print the final result print on console:
   //Gda⭐sk U⭐iversity of Tech⭐ology

3. Create a string containing your first and last name. Then revert it and print in the following format:
   //Jan Kowalski -> ikslawoK naJ

Control Flow (1 point)

In Swift you can use **if** and **switch** to make conditionals and **for-in**, **while** and **repeat-while** to make loops.

In the following example, an array containing six values is created. Then using for-in loop, the values of all elements are checked and if a value is larger than 50, then 5 points are added to additionalPoint, otherwise only 2 points are added:

```
let myScores = [30, 40, 45, 55, 60, 80]
var additionalPotins = 0

for score in myScores {
    if score > 50 {
        additionalPotins += 5
    } else {
        additionalPotins += 2
    }
}

print(additionalPotins)
//21
```

You can use **if** and **let** together to work with values that might be missing. These values are represented as optionals. An optional value either contains a value or contains *nil* to indicate that a value is missing. Write a question mark (**?**) after the type of a value to mark the value as optional.

```
var optionalName: String? = "Jan Kowalski"
var greeting = "Hello!"

if let name = optionalName {
    greeting = "Hello, \(name)"
```

```
}

print(greeting)
//Hello, Jan Kowalski
```

Feel free to set nil as a value of *optionalName*, to check what will happen. You can also add *else* clause to set a different string value.

Another way of handling optional values is to provide a default value using ?? operator. If the operator value is missing, the default value is used instead.

```
let firstName: String? = nil
let lastName: String = "Kowalski"
print("Hi \(firstName ?? lastName)")
//Hi Kowalski
```

In **switch** you can use any kind of data, for example string:

```
let icon = "☺️"

switch icon {
case "☺️":
   print("smiling face")

case "🐶":
   print("dog face")

case "🌵":
   print("cactus")


case "🎾":
   print("tennis ball")

default:
   print("No icon detected.")
}
```

Swift provides two kinds of for-in loop. You probably noticed the first one when checking the score in myScore array, where all elements in collection were checked. The second type is shown below, where two range operators are provided:

```
for i in 1...3 {
    print(i)
}
// print 1 - 3


for i in 1..<5 {
    print(i)
}
/ /print 1 - 4
```

To loop over a range in reversed order you can use the **reversed** method:

```
for i in (1...3).reversed() {
    print(i)
}
// print 3 - 1
```

When you would like to just execute something multiple times you can replace **i** with _

```
for _ in 1...5 {
    print("Hello Swift!")
}
// print “Hello Swift!” 5 times
```

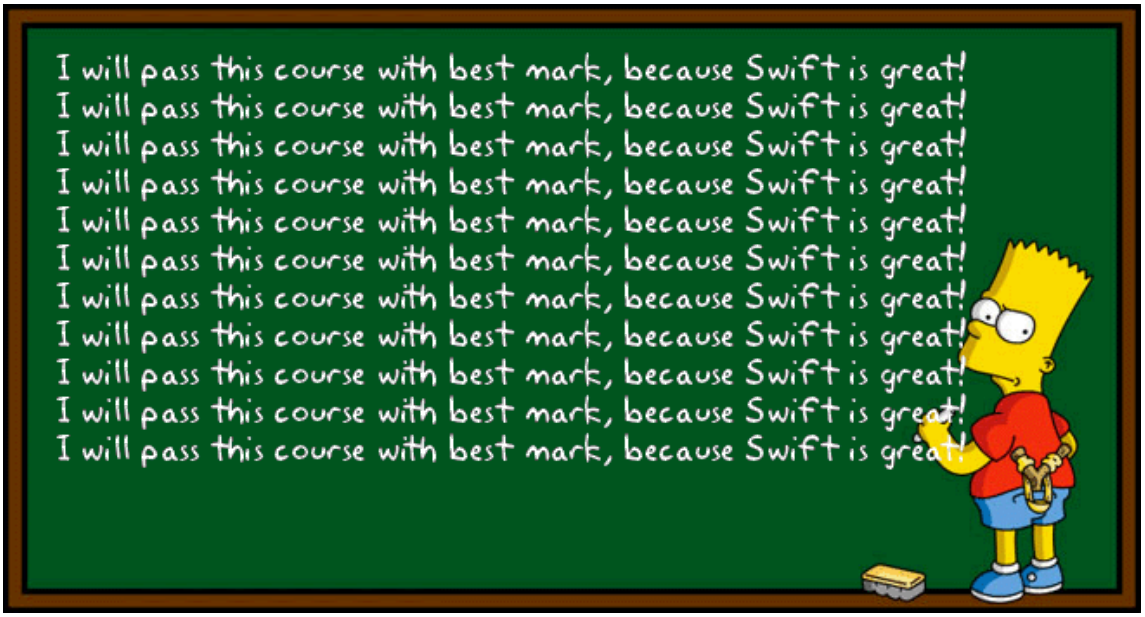
Use **while** to repeat a block of code until a condition changes. The condition of a loop can be at the end instead, ensuring that the loop is run at least once.

```
var n = 12
while n < 10 {
    n += 3
}
print(n)

var m = 12
repeat {
    m += 3
} while m < 10
print(m)
```

Exercise:

1. Print “I will pass this course with best mark, because Swift is great!” 11 times:



   but be smarter than Bart Simpson and use either **for** or **while** loop to present your results.

2. Print the first N square numbers. Example for number 5: (1*1 = **1** ; 2*2 = **4** ; 3*3 = **9** ; 4*4 = **16** ; 5*5 = **25**). Print just bold numbers.
3. “Draw” a square of N x N “@” sign. Example for N = 4:
   @@@@
   @@@@
   @@@@
   @@@@
   Tip: You can print multiple values in the same line by using a different terminator: print("@", terminator: "")

Collection Types

Swift provides three primary *collection types*, known as **arrays**, **sets** and **dictionaries**, for storing collections of values. Arrays are ordered collections of values. Sets are unordered collections of unique values, dictionaries are unordered collections of key-value associations.
Arrays, sets and dictionaries in Swift are always clear about the types of values and keys that they can store. This means that you cannot insert a value of the wrong type into a collection by mistake. It also means you can be confident about the type of values you will retrieve from a collection.

If you create an array, a set or a dictionary and assign it to a variable (**var**), the collection that is created will be *mutable*. This means that you can change (or mutate) the collection after it's created by adding, removing or changing items in the collection. If you assign an array, a set or a dictionary to a constant (**let**), that collection is *immutable* and its size and contents cannot be changed.

Arrays (1 point)

To create an empty array, you can do it using:

```
var arrayOfInt = [Int]()
//or
var arrayOfInt = Array<Int>()
```

These two forms are functionally identical and mean that you create a mutable array, which will contain Int type content. You can add values to array:

```
arrayOfInt.append(5)
```

Then print number of elements:

```
print("Array have \(arrayOfInt.count) elements.")
//Array has 1 elements.
```

And set it as empty array, but is still of type [Int]:

```
arrayOfInt = []
```

You can also initialize an array with an *array literal*, which is a shorthand way to write one or more values as an array collection. The example below creates an array called *students* to store **String** values:

```
var students: [String] = ["Hommer", "Lisa", "Bart"]
```

Thanks to Swift's type interface, you don't have to write the type of the array if you're initializing it with an array literal containing values of the same type. The initialization of *students* could have been written in a shorter form instead:

```
var students = ["Hommer", "Lisa", "Bart"]
```

Because all values in the array literal are of the same type, Swift can infer that [String] is the correct type to use for the *students* variable.

Use the Boolean isEmpty property as a shortcut for checking whether the count property is equal to 0, otherwise print all array contents:

```
if students.isEmpty {
    print("No students on board😞")
} else {
    print("\(students)")
}
```

You can add a new item to the end of an array:

```
students.append("Marge")
```

Alternatively, append an array of one or more compatible items with the addition assignment operator (**+=**).

```
students += ["Apu", "Barney", "Nelson"]
```

To get the value from array, which is stored at specific index you can use:

```
let firstStudent = students[0]
//Hommer
```

To change an existing value at a given index:

```
students[0] = "Flanders"
```

To remove an item from the array at a given index:

```
students.remove(at: 0)
```

Exercise:

1. Print the maximum value in array:
   `var numbers = [5, 10, 20, 15, 80, 13]`

2. Print the numbers from array in reversed order:
   `var numbers = [5, 10, 20, 15, 80, 13]`
   print: 13, 80, 15, 20, 10, 5

3. Create a new array containing unique numbers from *allNumbers* and then print it:
   `var allNumbers = [10, 20, 10, 11, 13, 20, 10, 30]`
   expect: unique = [10, 20, 11, 13, 30]

Sets (1 point)

To create an empty array, you can do it using:

```
var letters = Set<Character>()
```

To add value to set, use:

```
letters.insert("A")
```

You can remove all elements, but is still of type Set<Character>:

```
letters = []
```

The example below creates a set called *musicTypes* to store **String** values:

```
var musicTypes: Set<String> = ["Rock", "Classic", "Hip hop"]
```

Same like it was with arrays, Swift can recognize the type of Set, so:

```
var musicTypes: Set= ["Rock", "Classic", "Hip hop"]
```

To add new element to set use:

```
musicTypes.insert("Jazz")
```

To remove element:

```
musicTypes.remove("Hip hop")
```

To check whether a set contains a particular item, use the contains(_:) method:

```
if musicTypes.contains("Funk") {
    print("Is it your favourite?")
} else {
    print("Still something new to discover.")
}
```

You can efficiently perform fundamental set operations, such as combining two sets together, determining which values two sets have 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 with only the values common to both sets.
- Use the *symmetricDifference(_:)* method to create a new set with values in either set, but not both.
- Use the *union(_:)* method to create a new set with all of the values in both sets.
- Use the *subtracting(_:)* method to create a new set with values 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]

singleDigitPrimeNumbers.intersection(oddDigits).sorted()
//[3, 5, 7]

oddDigits.subtracting(singleDigitPrimeNumbers).sorted()
//[1, 9]

oddDigits.symmetricDifference(singleDigitPrimeNumbers).sorted()
//[1, 2, 9]
```

Exercise:

1. Find all divisors of *number* and store it in a set called *divisors*. Print it sorted.
   Input:
   `var number = 10`
   Output:
   divisors = [1, 2, 5, 10]

Dictionaries (1 point)

As with **arrays**, you can create an empty **Dictionary** of a certian type by using initializer syntax:
`var namesOfIntegers = [Int: String]()`

This example creates an empty dictionary of type **[Int: String]** to store human-readable names of integer values. Its keys are of type **Int**, and its values are of type **String**.

To add value to dictionary, use:

```
namesOfIntegers[16] = "sixteen"
```

You can remove all elements, but is still of type [Int: String]

```
namesOfIntegers = [:]
```

The example below creates a dictionary with type of [String: String] and a set two values to it:

```
var airports: [String: String] = ["GDN" : "Gdansk", "NYO": "Stockholm Skavsta"]
```

As with arrays, you don't have to write the type of the dictionary of you're initializing it with a dictionary litera whose keys and values have consistent types:

```
var airports = ["GDN" : "Gdansk", "NYO": "Stockholm Skavsta"]
```

You can add a new item to a dictionary with subscript syntax. Use a new key of the appropriate type as the subscript index, and assign a new value of the appropriate type:

```
airports["SFO"] = "San Francisco Airport"
```

You can use subscript syntax to change the value associated with a particular key:

```
airports["SFO"] = "San Francisco"
//["NYO": "Stockholm Skavsta", "SFO": "San Francisco", "GDN": "Gdansk"]
```

You can remove a key-value pair from a dictionary by assigning a value of **nil** for that key:

```
airports["SFO"] = nil
```

or use:

```
airports.removeValue(forKey: "SFO")
```

You can iterate over the key-value pairs in a dictionary with a for-in loop:

1.
```
for (airportCode, airportName) in airports {
    print("\(airportCode): \(airportName)")
}
//NYO: Stockholm Skavsta
//SFO: San Francisco
//GDN: Gdansk
```
2.
```
for airportCode in airports.keys {
    print("Airport code: \(airportCode)")
}
//Airport code: NYO
//Airport code: SFO
//Airport code: GDN
```
3.
```
for airportName in airports.values {
    print("Airport name: \(airportName)")
}
//Airport name: Stockholm Skavsta
//Airport name: San Francisco
//Airport name: Gdansk
```

Exercise:

1. You are given an array of dictionaries. Each dictionary in the array contains 2 keys *flightNumber* and *destination*. Create a new array of strings called *flightNumbers* that will contain only *flightNumber* value from each dictionary.
   ```
   var flights: [[String: String]] = [
      [
         "flightNumber" : "AA8025",
         "destination" : "Copenhagen"
      ],
      [
         "flightNumber" : "BA1442",
         "destination" : "New York"
      ],
      [
         "flightNumber" : "BD6741",
         "destination" : "Barcelona"
      ]
   ]
   ```
   expected: flightNumbers = ["AA8025", "BA1442", "BD6741"]

2. You are given an array of strings (names). Create new array of dictionaries which contains 2 keys *firstName*, which will be name from *names* array and *lastName* which will be hardcoded string value.
   ```
   var names = ["Hommer","Lisa","Bart"]
   ```
   expect: fullName = [
   ["lastName": "Simpson", "firstName": "Hommer"],
   ["lastName": "Simpson", "firstName": "Lisa"],
   ["lastName": "Simpson", "firstName": "Bart"]]

Conclusion

Congratulations 🙌🍾🏆🎉! You have finished your first Swift challenge. Now you know how to work with strings, loops and collections. Great job! Hope you will enjoy next exercise 👩‍💻👨‍💻.

Bibliography

- ”Excerpt From: Apple Education. “App Development with Swift.” Apple Inc. - Education, 2016. iBooks.
- ”The Swift Programming Language. Swift 4.0.3 Edition” Apple Inc. - Education, 2014. iBooks.