GDAŃSK UNIVERSITY OF TECHNOLOGY

Systemy Operacyjne macOS i iOS

Lab 2

More Swift!

Gdańsk 2021
Tomasz Idzi

Introduction

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

Functions (1 point)

Functions are self-contained chunks of code that perform a specific task. You give a function a name that identifies what it does, and this name is used to “call” the function to perform its task when needed.

When you define a function, you can optionally define one or more named, typed values that the function takes as input, known as *parameters*. You can also optionally define a type of value that the function will pass back as output when it is done, known as its *return type*.

The general syntax for a function is:

```
func name (list of parameter) -> return type {
        statements
}
```

Super-easy, huh? Don't waste time, implement your first funcy func in Playground 👩‍💻👨‍💻.

Exercises:

1. Write a function named **minValue** which takes two parameters of **Int** type and returns the smaller one. Try this function with a few values.
2. Write a function that takes an **Int** and returns it’s last digit. Use _ to ignore the external parameter name.
3. Implement the following functions. The **divides** function returns **true** if **a** is divisible by **b** and **false** otherwise. The **countDivisors** function should use the **divides** function to return the number of divisors of **number**. The **isPrime** function should use **countDivisors** function to determine if **number** is prime. Example of use:

```
divides(7, 3) // false - 7 is not divisible by 3
divides(8, 4) // true - 8 is divisible by 4

countDivisors(1) // 1 - 1
countDivisors(10) // 4 - 1, 2, 5 and 10
countDivisors(12) // 6 - 1, 2, 3, 4, 6 and 12

isPrime(3) // true
isPrime(8) // false
isPrime(13) // true
```

Closures (2 point)

Closures are self-contained blocks of functionality that can be passed around and used in your code. Closures can capture and store references to any constants and variables from the context in which they are defined. Closure expression syntax has the following general form:

```
{ (parameters) -> return type **in**
        statements
}
```

- The closures that don't have parameters and return *String* value:

```
var helloWorld: () -> (String) = {
   return "Hello World!"
}

helloWorld() // Hello World!
```

- The closures that take two Int parameters and return Int value:

```
var addTwoInt: (Int, Int) -> (Int) = { x, y in
   return x + y
}

addTwoInt(2,3) // 5
```

- We have possibilities to set closures to other variables:

```
var add = addTwoInt
add(3,7) // 10
```

- The closures can also be a parameter of a function, when it's the last one we can write it after function call:

```
func sumOfFuncDefine(a:Int, b:Int, f: (Int)->(Int)) -> Int {
   var sum = 0
   for i in a...b {
      sum += f(i)
   }
   return sum
}
```

```
sumOfFuncDefine(a: 1, b: 5) {
    $0
} // Return sum of the first 5 numbers

sumOfFuncDefine(a: 1, b: 5) { (a) -> (Int) in
    return a * a
}// Return sum of the first 5 squares
```

In Swift we have access to lot of useful functions using closures, especially when working with arrays:

```
var array = [10, 20, 30, 2, 32, 120, 1, 3, 34]
```

- **Sort** - need to define condition of sorting array:

  ```
  array.sorted{ $0 < $1 }
  //[1, 2, 3, 10, 20, 30, 32, 34, 120]
  ```

- **Map** - loop over a collection and apply the same operation to each element in the collection:

  ```
  array.map{"\($0)"}
  //["10", "20", "30", "2", "32", "120", "1", "3", "34"]
  ```

  instead of:

  ```
  var strArray: [String] = []
  for number in array {
      strArray.append("\(number)")
  }
  ```

- **Filter** - loop over a collection and return an **Array** containing only those elements that match an include condition:

  ```
  array.filter{$0 > 10}
  //[20, 30, 32, 120, 34]
  ```

- **Reduce** - combine all items in a collection to create a single new value. In other words, reduce take two parameters, an initial value and a closure that will be used to combine the elements of the array:

  ```
  array.reduce(0){$0 + $1}
  //252

  let codes = ["abc","def","ghi"]
  ```

```
let text = codes.reduce("123_"){$0 + $1}
//123_abcdefghi
```

Exercises:

1. Do you remember Bart Simpson from the last lab? Let's do the same thing as the last time, but now with a closure, write a function named **smartBart** that takes an integer **n** and **closure** and calls the closure **n** times. The closure will not take any parameters and will not have a return value. Try it with: "I will pass this course with best mark, because Swift is great!"
2. Use **filter** to create an array that contains all the multiples of 4 from array:

   ```
   let numbers = [10, 16, 18, 30, 38, 40, 44, 50]
   ```

3. Find the largest number from the **numbers** array and then print it. Use **reduce** to solve this exercise.
4. Join all the strings from **strings** into one using **reduce.** Add spaces in between strings. Print your result.

   Input:
   ```
   var strings = ["Gdansk", "University", "of", "Technology"]
   ```
   Output:
   ```
   Gdansk University of Technology
   ```

5. Find the sum of the squares of all the odd numbers from **numbers** and then print it. Use **map**, **filter** and **reduce** to solve this problem.

   Input:
   ```
   let numbers = [1, 2 ,3 ,4, 5, 6]
   ```
   Output:
   ```
   1 + 9 + 25 -> 35
   ```

Tuples (1 point)

Tuples group multiple values into a single compound value. The values within a tuple can be of any type and don't have to be of the same type as each other.

```
let http404Error = (404, "Not Found")
```

```
http404Error.0  //404
http404Error.1 //Not Found
```

Exercises:

1. Write a function named **minmax** that takes two integers and returns both the minimum and the maximum values inside a tuple.
2. You are given an array of strings stored in the variable **strings**. Create a new array named **countedStrings** containing values of type **(String, Int)**. Each tuple contains a string from the **strings** array followed by an integer indicating how many times it appears in the **strings** array. Each string should only appear once in the **countedStrings** array.

   Input:

   ```
   var stringsArray = ["gdansk", "university", "gdansk", "university", "university",
   "of”,  "technology", "technology", "gdansk", "gdansk"]
   ```

   Output:

   ```
   //[("gdansk", 4), ("university", 3), ("of", 1), ("technology", 2)]
   ```

Enums (1 point)

Use *enum* to create enumeration. Like classes and all other named types, enumerations can have methods associated with them.

```
enum Device: String {
   case iPhone4
   case iPhone5
   case iPhone6
   case iPhone7
   case iPhone8
   case iPhoneX

   func diagonalDescription() -> String {
      switch self {
      case .iPhone4:
         return "3.5\""
      case .iPhone5:
         return "4.0\""
      case .iPhone6, .iPhone7, .iPhone8:
         return "4.7\""
      case .iPhoneX:
         return "5.8\""
```

```
        }
    }
}

let device = Device.iPhoneX
device.diagonalDescription() //5.8"
```

Exercises:

1. Create an enum with name **Day**, which will contain cases of all weekdays. Every **rawValue** of the case should represent a day number during the week (Monday = 1, Tuesday = 2, etc.). Create an additional func which will display emoji for specific day.

Conclusion

Congratulations 🙌🍾🏆🎉! You have finished your another Swift challenge. Now you know how to work with functions, closures, tuples, enums, classes and structures. Great job! Hope you will enjoy the 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.