Created by: Zoltán Richárd Jánki
Date: 03.03.2017

Apple Swift Course
Practice 3

Collections:

Array:

- `let array = []` - empty array constant
- `var array = []` - empty array variabe
- `var array = [String]()` - empty String Array defined by the initialization method

e.g.: `let toDoList = [ "Breakfast", "Lunch", "Dinner" ]`
`var toDoList2 = [ "Breakfast", "Lunch", "Dinner" ]`
`toDoList2.append("Snack")` - add a new element (push)
`toDoList2.count` - the number of the elements in the array

Set:

- `var set = Set<String>()` - empty set of strings defined by the initialization method

e.g.: `var listSet: Set<String> = [ "Bread", "Milch", "Salad" ]`
`listSet.insert("Watermelon")`

Dictionary:

collection defined by key-value pairs

- `var intStrDictionary = [Int: String]()` - empty dictionary created by the initialization method, where the key is `Int`, the value is `String`

e.g.: `var priceList = [ "Bread" : 250, "Milch" : 220, "Salad" : 350 ]`

Tuple:

- use more values at the same time (for passing, returning, comparing, etc...)
- we use it in quasi vector-form

e.g.: `let tuple1 = (10, 20, 30, 40)`
`let tuple2 = (20, 10, 30, 40)`

Functions:

- *basic function syntax:*

```
func funcName(param1:param1Type, param2:param2Type) -> returnType {}
```

`funcName`: the name of the function
`param1, param2`: the name of the parameters
`param1Type, param2Type`: the types of the parameters
`-> returnType`: the return value of the function

e.g.:
```
func greeting(str: String) → String {
    return str        //a function that needs a String parameter, and returns a String as well
}

let helloString = "Hello World!"
greeting(helloString)
```

- function with argument label:

```
func funcName(extParam intParam: paramType) -> returnType {}
     extParam: argument label (use it outside of the scope, e.g.: calling this function)
     intParam: internal parameter name (use it inside of the function scope)

e.g.: func multiply(extInt intInt: Int) -> Int {
     return intInt * 10   //we use the internal parameter name (intInt)
}

multiply(extInt: 10)      //if we call the function, we use the argument label (extInt)
```

- function with default parameter:

```
func funcName(param: paramType = defaultValue) -> returnType {}

e.g.: func multiply(myInt: Int = 100) -> Int {
     return myInt * 10    //the default value of myInt is 100
}

multiply(200)   //the return value is 2000
multiply()      //the return value is 1000 (because the default is 100)
```

- function with variadic parameters:

```
func funcName(param: paramType...) -> returnType {}

e.g.: func varParam(numbers: Double...) -> Int {
     return numbers.count
}

varParam(10.0,20.1,0.5,2.66665)     //return value: 4
```

- nested functions:

```
func funcName(param: paramType) -> (nestedParam: paramType) ->
returnType {}

e.g.: func nestedIncrease(myInt: Int) -> (Int) -> Int {
     func increase(number: Int) -> Int { return number + 1 }
     func decrease(number: Int) -> Int { return number - 1 }

     if (myInt > 20) {
          return increase       //return value is a function
     } else {
          return decrease       //return value is a function
     }
}

var myScore = 500
let increaser = nestedIncrease(21)    //21 > 20, so the increase will be returned
increaser(myScore)                    //500 + 1 = 501 will be returned
```

- *functions with in-out parameter:*
- változtatható a paraméter értéke

```
func funcName(param: inout paramType) -> returnType {}
```

```
e.g.: var myInt = 10
func inOutFunc(int: Int) -> Int {
      return int *= 10
}

inOutFunc(myInt)        //return value: 100
```

Closure:

- it's the special case of nested functions
- nested functions that pass value
- name isn't defined

- *closure syntax:*

```
{(param1: param1Type, param2: param2Type) -> returnType in return
returnStatement}
```

```
e.g.: var names = ["John", "Carlos", "Sylvester", "Andrew"]

var reversedNames = names.sort({(str1: String, str2: String) ->
Bool in return str1 > str2})    //decreasing order
```

- *closure that changes the positions of the first and the last element:*

```
e.g.: var changeOrder: () -> [String] = {
      if let a = names.first {
            names[0] = names[names.count-1]
            names[names.count-1] = a
      }
      return names
}
```

Control flows:

if:

```
if (condition) {
      statement1
} else {
      statement2
}
```

for:

```
let ten = 10
for i in 1...ten {      // the i variable only exists in the for loop's scope
      print(i)          // the i goes through the elements and its value is the actual element
}

for index in 1...5 {
      print(index)      //index variable goes from 1 to 5
}
```

```
for index in 1..<5 {
    print(index)     //index variable goes from 1 to 4 (just .. can be written, if there's no
}                    //equation!)

for index in 1..<5.reverse() {
    print(index)     //cycle the goes backwards (index goes from 4 to 1)
}

for index in 0.stride(to: 10, by: 2) {
    print(index)     //the index variable goes from 0 to 10 in steps of 2
}
```

for (if the variable and its value isn't used):

```
var solution = 1
for _ in 1...ten {              //if the cycle-variable isn't needed
    solution += 1               //can be signed with _
}
```

while:

```
var solution = 0
while(solution < 10) {          //simple while-loop
    solution += 10              //before the statement there's a check!!!
    print("\(solution)")
}
```

repeat-while:

```
var solution = 0
repeat {                        //the well-known do-while cycle in Swift
    solution += 10              //the statement runs before the check of the condition
    print("\(solution)")        //the value of solution will be 10
} while (solution < 0)
```

switch:

```
var solution = 10
switch solution {
    case let x where x < 50: print("lower")
    case let x where x > 50: print("higher")
    default: print("exactly 50")
}           //there are no break statements among the cases
            //we can define new variable inside of the scope (x)
            //the new variable can be checked with the where condition
```

guard (early exit):

```
let x = 10
guard x == 10 else {        //if the condition right after the guard isn't accomplished,
    return                  //the else part will be only executed, and nothing else inside of
}                           //the scope
print("It was ten.")
```

Operators:

Assignment	=
Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulo (integer remainder)	%
Equal to	==
Not equal to	!=
AND	&&
OR	||
NOT (Logical negation)	!
Nil-coalescing	??
If-then-else (ternary)	? :

- The evaluation of the logical operátors occurs from **left** to **right**.
- The assignment of Optional has a very elegant method: the **Nil-coalescing** operator (**??**)

```
e.g.: var e: Int?
var f  = 20
e != nil ? e! : f       //traditional solution
var sol = e ?? f        //elegant solution in Swift (if e is not nil, then the values can be
                        // passed, else f will be passed)
```

Structs:

- defined by the keyword **struct**

e.g.: **struct myStruct {}**

Struct vs. **Class**		
Similarities	**Differences**	
	Struct	**Class**
Can have attributes	No inheritance	Inheritance works
Can have functions	Has no **deinit()** function	**deinit()** function
Have an initializer method	No reference counting (no ARC)	ARC makes possible the have more than one references
Can be extended	It will be copied if it is passed	
Can make protocol to them		

Initialization:

- occurs by the call of **init()** function at both classes and structs
- can be parameterized arbitrarily
- the Optionals have to be always initialized
- a class can have more than one **init()** functions
- if no **init()** function is defined, then the basic **init()** function is called (without any parameters)

Enumeration:

- defined by the **`enum`** keyword
- cases are defined by the **`case`** keyword

e.g.:
```
enum Compass {
      case north
      case south
      case east
      case west
}
```

- reference: **`EnumName.enumCase`**

e.g.: `var myDirection = Compass.north`

- if the type is known, you can use the short form of the reference

e.g.: `myDirection = .east` //**`myDirection`** was declared in the **`Compass`** enumeration