Functions as first class citizens

first class citizens can:

- be stored in a variable or constant
- be passed as arguments to functions
- be returned by functions

// be stored in a variable

```
func getCourseName() -> String {
    return "App Design"
}

let courseNameFunction = getCourseName

print(courseNameFunction())
```

// be passed as arguments to functions

```
func course(name: () -> String) {
    print(name())
}
```

or ⇒

```
func course(name: () -> String) -> String {
    return (name())
}
```

```
course(name: getCourseName)
```

// be returned by functions

```
func getCourseNameFunction() -> () -> String {
    return getCourseName
}

getCourseNameFunction()()
```

// stored in some kind of data structure

```
let courseNameArray = [getCourseName, getCourseName, getCourseName]

for courseName in courseNameArray {
    print(courseName())
}
```

(or)

```
courseNameArray[0]()
```

*functions ⇒ self contained blocked of code

Closures as first class citizens

closures are basically just functions without names, they are unique that they are able to capture values from their surrounding context. Since closures are functions they behave as first class citizens

// assign to a constant or variable

```
func add() -> Void {
    func nestedAdd() {


    }
}
```

```
func addNumber(_ number1: Int, _ number2: Int) -> Int {
    return number1 + number2
}

let addNumbersClosure = { (_ number1: Int, _ number2: Int) -> Int in
    return number1 + number2
}
```

⇓ or

```
let addNumbersClosure: (Int, Int) -> Int = { number1, number2) -> Int in
    return number1 + number2
}

let multiplyNumbersClosure: (Int, Int) -> Int = { number1, number2) -> Int in
    return number1 * number2
}
```

// passing to a function

```
func add(numbers: (Int, Int) -> Int) {
    numbers(3, 2)
}

add(numbers: addNumbersClosure)

add { (num1, num2) -> Int in
    return num1 + num2
}
```

// return from a function

```
func addNumbersFunction() -> (Int, Int) -> Int {
    return {(number1, number2) in
        return number1 + number2
    }
}

let addNumbers = addNumbersFunction()
addNumbers(3, 4)
```

// closure expressions

General closure expression

```
{(parameters) -> (type) in
    return value
}
```

Inferring type from context

```
{ parameter in
    return value
}
```

Implicit Returns (for single expression closure

```
{ parameters in
    value
}
```

```
let addNumbersGeneralExpression = { (number1: Int, number2: Int) -> Int in
    return number1 + number2
}

let addNumbersInferredExpression : (Int,Int) -> Int = { (number1, number2) in
    return number1 + number2
}
let addNumbersInferredExpression = { (number1: Int, number2: Int ) in
    return number1 + number2
}

let addNumbersImplicitReturnExpression = { (number1: Int, number2: Int) in
    number1 + number2
}

let addNumbersShortHandExpression: (Int, Int) -> Int = { $0 + $1 }

func add (numbers: (Int, Int) -> Int){
    print (numbers (3, 2))
}
// general expression                    => trailing closure
add { (num1, num2) -> Int in
    return num1 + num2
}
// inferred type
add { (num1, num2) in
    return num1 + num2
}
// implicit return
add { (num1, num2) in
    num1 + num2
}
//shorthand argument type
add { $0 + $1 }


func getFullName ( firstName: String, lastName: String, result: (String) -> Void){
    let fullName = firstName + " " + lastName
    result (fullName)
}

getFullName (firstName: "Steve", lastName: "Jobs" ){ (fullName) in
    print (fullName)
}
```

Nescaping closure

```
func fetchData(completionHandler: @escaping (_ response: String?) -> Void) {
    guard let urlComponents = URLComponents(string: "http.....") else {
        completionHandler(nil)
        return
    }
    guard let url = urlComponent.url else{
        completionHandler(nil)
        return
    }
    var request = URLRequest(url: url)
    request.httpMethod = "GET"
    request.addValue("application/json; charset=utf-8", forHTTPHeaderField: "Accept")
    let config = URLSession.configuration.default
    let session = URLSession(configuration: config)
    let task = session.dataTask(with: request) { (data, response, error) in
        if let error = error {
            print(error.localizedDescription)
            completionHandler(nil)
            return
        }
        guard let data = data,
            let response = String(data: data, encoding: String.Encoding.utf8)else{
            completionHandler(nil)
            return
        }
        completionHandler(response)
        task.resume
    }
}

fetchData { (response) in
    if let response = response {
        print(response)
    }else{
      print("failed to get responce")
    }
}
```

Type Alias ⇒ a type alias is used to define an alternative name for an existing type

```
import Foundation
enum APIError: Error{
    case failedToGetResponse
}

typealias APIResponse = (_ response: String?) -> Void

// or  // come with swift 5
typealias APIResponse = (_ response: Result<String, APIError>) -> Void

func fetchData(completionHandler: @escaping APIResponse){
    guard let urlComponents = URLComponents(string: "http.....") else{
        completionHandler(.failure(APIError.failedToGetResponse))
        return
    }
    guard let url = urlComponent.url else{
        completionHandler(.failure(.failedToGetResponse))
        return
    }
    var request = URLRequest(url: url)
    request.httpMethod = "GET"
    request.addValue("application/json; charset=utf-8", forHTTPHeaderField: "Accept")
    let config = URLSessionConfiguration.default
    let session = URLSession(configuration: config)
    let task = session.dataTask(with: request){ (data, response, error) in
        if let error = error{
            print(error.localizedDescription)
            completionHandler(.failure(.failedToGetResponse))
            return
        }
        guard let data = data,
            let response = String(data: data, encoding: String.Encoding.utf8) else{
            completionHandler(.failure(.failedToGetResponse))
            return
        }
        completionHandler(.success(response))
        task.resume
    }
}
```

```
fetchData{(response) in
    if let response = response{
        print(response)
    }else{
        print("failed to get response")
    }
}
```

⇒

```
fetchData{(response) in
    switch response{
    case .success(let success):
        print(success)
    case .failure(let error):
        print(error.localizedDescription)
    }
}
```

// Higher order functions

⇒ takes one or more functions as arguments or return a function as its result

map - transform each element of an Array
filter - only returns the elements of an array that pass a certain condition
reduce - combines all the elements of an array into a single value
mapValues - transforms only the values of a dictionary
compactMap (New To Swift 5) - removes all elements in dictionary that are nil

⇒ return a new array / dictionary
Not mutating

```
import Foundation

let numberArray = [2, 4, 7, 5, 1, 9]
let simpleDictionary: [String: String?] = ["a": "1", "b": nil, "c": "3"]

let arrayOfStrings = ["Car", "Boat", "Plane", "Building"]
```

// without higher order function

```
var characterCountArray = [Int]()
for string in arrayOfStrings {
    characterCountArray.append(string.count)
}
```

⇒

// example with Map

```
let characterCountArray = arrayOfStrings.map { $0.count }
```

↓ or

```
let characterCountArray = arrayOfStrings.map { (string) -> Int in
    return string.count
}
```

// filter example

```
// 3.isMultiple(of: 2) ⇒ will return true or false
numberArray.filter { $0.isMultiple(of: 2) }
```

// reduce example

```
numberArray.reduce(0) { $0 + $1 }
```

// mapValues example

```
simpleDictionary.filter { $0.value != nil }.mapValues { $0! }
```

$0.key

make sure every thing not optional

⇒ before swift 5

// compactMap example

⇒ new in swift 5

```
simpleDictionary.compactMapValues { $0 }
```

// Functional Programming In Practice ⇒ AVOID mutability (use more constants possible)

// Pure Function ⇒ 1. always returns the same output given the same input
2. has no side-effects

```
var newResult = 0
func notPureFunction(num1: Int, num2: Int) {
    newResult = num1 + num2
}
```

```
func pureFunction(num1:Int, num2:Int) -> Int {
    return num1 + num2
}
let newResultFromPureFunction = PureFunction(num1: 2, num2: 3)
```

// Higher order functions

```
let studentScores = [3, 4, 1, 7, 2, 10, 5]
func nonFunctionalStudentScoreFilter(_ scores:[Int]) -> [Int] {
    var bestScores = [Int]()
    for score in scores {
        if score > 5 {
            bestScores.append(score)
        }
    }
    return bestScores
}
```

// right way :)

```
func functionalStudentScoreFilter(_ scores:[Int]) -> [Int] {
    return scores.filter { $0 > 5 }
}
```

// Declarative

```
let locationTemps = ["SA": 23, "JP": 10, "GR": 15, "BG": 25]
struct WarmCity {
    var cityName: String
    var temp: Int
}
var warmCities = [WarmCity]()

for locationTemp in locationTemps {
    if locationTemp.value > 16 {
        warmCities.append(WarmCity(cityName: locationTemp.key, temp: locationTemp.value))
    }
}
```

//

```
locationTemps.filter { $0.value > 16 }.map { (city, temp) -> WarmCity in
    WarmCity(cityName: city, temp: temp)
}
```