VII / Closures : part two

Using closures as parameters when they accept parameters

=> a closure you pass into a function can also accept its own parameters.

Exemple : **travel()** function that accepts a closure as its only parameter

Function syntaxe:

```
func travel(action: (String) -> Void) {
    print("I'm getting ready to go.")
    action("London")
    print("I arrived!")
}
```

Trailing closure syntax:

```
travel { (place: String) in
    print("I'm going to \(place) in my car")
}
```

Total score: 12/12 Checked

Using closures as parameters when they return values

=> Instead of () -> Void when a closure return nothing replace that Void with any type of data to force the closure to return a value.

Exemple 1 :

```
func travel(action: (String) -> String) {
    print("I'm getting ready to go.")
    let description = action("London")
    print(description)
    print("I arrived!")
}
```

=> Now when we call travel() using trailing closure syntax, our closure code is required to accept a string and return a string:

```
travel { (place: String) -> String in
    return "I'm going to \(place) in my car"
}
```

When to use closures with return values as parameters to a function?

For example, if we wanted to reduce the array **[10, 20, 30]**, it would work something like this:

- It would create a variable called **current** with a value set to the first item in its array. This is our starting value.
- It would then start a loop over the items in the array that got passed in, using the range **1...** so that it counts from index 1 to the end.
- It would then call the closure with 10 (the current value) and 20 (the second value in the array).
- The closure might be reducing the array using addition, so it would add 10 to 20 and return the sum, 30.
- Our function would then call the closure with 30 (the new current value) and 30 (the third item in the array).
- The closure would add 30 to 30 and return the sum, which is 60.
- Our function would then send back 60 as its return value.

Exemple 2 :

```
func reduce(_ values: [Int], using closure: (Int, Int) -> Int) -> Int {
    // start with a total equal to the first value

    var current = values[0]

    // loop over all the values in the array, counting from index 1 onwards

    for value in values[1...] {
        // call our closure with the current value and the array element, assigning its result to our current value
```

```
        current = closure(current, value)
    }

    // send back the final current value

    return current
}
```

– – – – – – –

```
let numbers = [10, 20, 30]

let sum = reduce(numbers) { (runningTotal: Int, next: Int) in
    runningTotal + next
}

print(sum)
```

_____

Exemple 3 :

```
func increaseBankBalance(start: Double, interestCalculator: () -> Double)
{
        print("Your current balance is \(start).")
        let interestRate = interestCalculator()
        let withInterest = start * interestRate
        print("You now have \(withInterest).")
}
increaseBankBalance(start: 200.0) {
        return 1.01
}
```

Total score: 12/12 checked

_____
_____

Shorthand parameter names

Exemple :

1.

```
func travel(action: (String) -> String) {
    print("I'm getting ready to go.")
    let description = action("London")
    print(description)
    print("I arrived!")
}

travel { (place: String) -> String in
    return "I'm going to \(place) in my car"
}
```

– – – – – – – – –

2. => Swift *knows* the parameter to that closure must be a string, so we can remove it:

```
travel { place -> String in
    return "I'm going to \(place) in my car"
}
```

– – – – – – – — –

3. => It also knows the closure must return a string, so we can remove that:

```
travel { place in
    return "I'm going to \(place) in my car"
}
```

– – — – — – – – –

4. => As the closure only has one line of code that must be the one that returns the value, so Swift lets us remove the **return** keyword too:

```
travel { place in
    "I'm going to \(place) in my car"
}
```

- - - - - - - - - - -

5. => Swift has a shorthand syntax that lets you go even shorter. Rather than writing **place in** we can let Swift provide automatic names for the closure's parameters. These are named with a dollar sign, then a number counting from 0:

```
travel {
    "I'm going to \($0) in my car"
}
```

Total score: 6/6 checked

Closures with multiple parameters

=> This time our travel() function will require a closure that specifies where someone is traveling to, and the speed they are going. This means we need to use (String, Int) -> String for the parameter's type:

Exemple :

```
func travel(action: (String, Int) -> String) {
    print("I'm getting ready to go.")
    let description = action("London", 60)
    print(description)
    print("I arrived!")
}
```

- - - - - - - — -

=> using a trailing closure and shorthand closure parameter names

```
travel {
    "I'm going to \($0) at \($1) miles per hour." <- $0 = first parameter / $1
second parameter
}
```

Total score: 12/12 checked

Returning closures from functions

=> he same way that you can pass a closure *to* a function, you can get

closures returned *from* **a function too.**

- NB: The syntax for this is a bit confusing a first, because it uses **->** twice: once to specify your function's return value, and a second time to specify your closure's return value.

- NB: Function that accepts no parameters, but returns a closure. The closure that gets returned must be called with a string, and will return nothing

Exemple :

```
func travel() -> (String) -> Void {
    return {
        print("I'm going to \($0)")
    }
}
```

- - - - - — - - - -

=> call travel() to get back that closure, then call it as a function:

```
let result = travel()
result("London")
```

_____

=> The most common situation is effectively this: I need a function to call, but I don't know what that function is. I know how to *find out* **that function – I know who to** *ask* **to find the function – but I don't know myself.**

Exemple 1 : write a function that returned one random number between 1 and 10,

```
func getRandomNumber()-> Int {
    Int.random(in: 1...10)
}
```

=> That will return a random integer every time it's called

```
print(getRandomNumber())
```

_____

Exemple 2 : function that returns a closure that, when called, will generate a random number from 1 through 10

```
func makeRandomGenerator() -> () -> Int {
    let function = { Int.random(in: 1...10) }
    return function
}
```

- - - - - - -

=> inside the function, we create a closure that wraps **Int.random(in: 1...10)** and send back that closure

```
let generator = makeRandomGenerator()
let random1 = generator()
print(random1)
```

Total score: 12/12 checked

_____

Capturing values

=> use any external values inside your closure, Swift *captures* **them – stores them alongside the closure, so they can be modified even if they don’t exist any more.**

Exemple 1:

```
func travel() -> (String) -> Void {
    return {
        print("I'm going to \($0)")
    }
}
```

- - - - - - - -

=> to get back the closure, then call that closure freely:

```
let result = travel()
result("London")
```

- - - - - - - - - -

=> Closure capturing happens if we create values in **travel()** that get used inside the closure. For example, we might want to track how often the returned closure is called:

```
func travel() -> (String) -> Void {
    var counter = 1

    return {
        print("\(counter). I'm going to \($0)")
        counter += 1
    }
}
```

=> it gets captured by the closure so it will still remain alive for that closure.

```
result("London")
result("London")
result("London")
```

Check web site: https://alisoftware.github.io/swift/closures/2016/07/25/closure-capture-1/

_____

Exemple 2 :

```
func visitPlaces() -> (String) -> Void {
    var places = [String: Int]()
    return {
        places[$0, default: 0] += 1
        let timesVisited = places[$0, default: 0]
        print("Number of times I've visited \($0): \(timesVisited).")
    }
```

```
}
let visit = visitPlaces()
visit("London")
visit("New York")
visit("London")
```

Exemple 3 :

```
func summonGenie(wishCount: Int) -> (String) -> Void {
    var currentWishes = wishCount
    return {
        if currentWishes > 0 {
            currentWishes -= 1
            print("You wished for \($0).")
            print("Wishes left: \(currentWishes)")
        } else {
            print("You're out of wishes.")
        }
    }
}
let genie = summonGenie(wishCount: 3)
genie("a Ferrari")
```

Total score: 12/12 checked