V / Functions

=> functions start with the func keyword

Exemple :

```
func printHelp() {
    let message = """
Welcome to MyApp!

Run this app inside a directory of images and
MyApp will resize them all into thumbnails
"""

    print(message)
}

printHelp()
```

Total score: 12/12 checked

Accepting parameters

=> functions accept parameters, give each parameter a name, then a colon, then tell Swift the type of data it must be

Exemple :

```
func square(number: Int) {
    print(number * number)
}

square(number: 8) => return 64
```

Total score: 12/12 checked

———

Returning values

=> functions can also send back data and then tell Swift what kind of data will be returned.

Exemple :

```
func square(number: Int) -> Int {
    return number * number
}

let result = square(number: 8)
print(result) => printed 64
```

———————————————————

=> Two ways to send back multiple pieces of data

- Using a tuple, such as **(name: String, age: Int)**
- Using some sort of collection, such as an array or a dictionary.

Exemple 1 :

```
func getUser() -> [String: String] {
    ["first": "Taylor", "last": "Swift"]
}

let user = getUser()
print(user["first"])
```

———————————

Exemple 2 :

```
func getUser() -> (first: String, last: String) {
    (first: "Taylor", last: "Swift")
}
```

```
let user = getUser()
print(user.first)
```

Total score: 12/12 checked

Parameter labels

=> Swift lets provide two names for each parameter:

- one to be used externally when calling the function
- one to be used internally inside the function

Exemple 1 :

```
func sayHello(to name: String) {
    print("Hello, \(name)!")
}


sayHello(to: "Taylor")
```

Exemple 2:

```
func setReactorStatus(primaryActive: Bool, backupActive: Bool,
isEmergency: Bool) {
    // code here
}

setReactorStatus(primaryActive: true, backupActive: true, isEmergency:
false)
```

Exemple 3 :

```
func setAge(for person: String, to value: Int) {
    print("\(person) is now \(value)")
```

```
}

setAge(for: "Paul", to: 40)
```

Exemple 4:

```
func numberOfTims(in array: [String]) -> Int {
	var count = 0
	for name in array {
		if name == "Tim" {
			count += 1
		}
	}
	return count
}
```

Exemple 5 :

```
func countDown(from start: Int) {
	var current = start
	while current >= 0 {
		print("\(current)...")
		current -= 1
	}
}
countDown(from: 10)
```

NB : externally it’s called **to**, but internally it’s called **name**

Total score: 12/12 checked

---

omit a parameter label

=> Reason for skipping a parameter name is when your function name is a verb and the first parameter is a noun the verb is acting on

- Greeting a person would be **greet(taylor)** rather than **greet(person:**

taylor)

- Buying a product would be **buy(toothbrush)** rather than **buy(item: toothbrush)**
- Finding a customer would be **find(customer)** rather than **find(user: customer)**

=> When the parameter label is likely to be the same as the name of whatever you're passing in

- Singing a song would be **sing(song)** rather than **sing(song: song)**
- Enabling an alarm would be **enable(alarm)** rather than **enable(alarm: alarm)**
- Reading a book would be **read(book)** rather than **read(book: book)**

Total score: 12/12 checked

Default parameters

=> give parameters a default value just by writing an = after its type followed by the default you want to give it

Exemple 1:

```
func greet(_ person: String, nicely: Bool = true) {
    if nicely == true {
        print("Hello, \(person)!")
    } else {
        print("Oh no, it's \(person) again...")
    }
}

greet("Taylor")
greet("Taylor", nicely: false)
```

Exemple 2 :

```
func findDirections(from: String, to: String, route: String = "fastest",
avoidHighways: Bool = false) {
    // code here
}

findDirections(from: "London", to: "Glasgow")
findDirections(from: "London", to: "Glasgow", route: "scenic")
findDirections(from: "London", to: "Glasgow", route: "scenic",
avoidHighways: true)
```

NB: Shorter, simpler code most of the time, but flexibility when we need it

Total score: 12/12 checked

Variadic functions

=> make any parameter variadic by writing ... after its type
Swift converts the values that were passed in to an array of integers

Exemple 1 :

```
func square(numbers: Int...) {
    for number in numbers {
        print("\(number) squared is \(number * number)")
    }
}

square(numbers: 1, 2, 3, 4, 5)
```

Exemple 2 :

open() function that opened a file for editing in Preview

```
open("photo.jpg")

open("photo.jpg", "recipes.txt", "myCode.swift")
```

Total score: 6/6 checked

Writing throwing functions

=> throw errors from functions by marking them as throws before their return type

Exemple :

```
enum PrintError: Error {
	case invalidCount
}
func printPages(text: String, count: Int) throws {
	if count <= 0 {
		throw PrintError.invalidCount
	} else {
		for _ in 1...count {
			print("Printing \(text)...")
		}
	}
}
```

Total score: 12/12 checked

Running throwing functions

!! Swift won't let run an error-throwing function by accident !!

=> If any errors are thrown inside the do block, execution immediately jumps to the catch block. Let's try calling checkPassword() with a parameter that throws an error:

```
do {
   try checkPassword("password")
   print("That password is good!")
```

```
} catch {
    print("You can't use that password.")
}
```

NB : Use **do** to start a section of code that calls throwing functions
=> If any errors are thrown, execution immediately jumps to the **catch** block.
=> Throwing functions must be marked with **throws**
=> Throwing functions must be called using **try**

Total score: 6/6 checked

---

inout parameters

!! All parameters passed into a Swift function are *constants !!*

=> Excepte if we add inout beside the type of the parameter for the declaration of the function
=> Inout parameters must be passed in using &

Exemple :

```
func doubleInPlace(number: inout Int) {
    number *= 2
}

var myNum = 10
doubleInPlace(number: &myNum)
```

NB : Changing Inout parameters inside a function changes them outside too.

Total score: 6/6 checked