SWIFT INTRO :
FUNCTIONS & ERROR HANDLING

✅ Functions:
Using func

```
func functionName() {

  // function content
}

//calling function
functionName()
```

- Function with parameter:

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

call function

```swift
square(number: 8)
```

NOTE: a function could take no parameters or **take 20** of them

✅ Function Return values:

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

+ Swift lets us skip using the **return** keyword when we have **ONLY 1 expression** in our function.

```swift
//Skipping 'return' if function only have 1 single expression

func addNum (num1 : Int, num2 : Int) -> Int {
    num1 + num2 //no 'return' here!
}

print(addNum(num1: 5, num2: 7))
```

```
12

"12\n"
```

IMPORTANT:

```swift
func greet(name: String) -> String {
    if name == "Taylor Swift" {
        return "Oh wow!"
    } else {
        return "Hello, \(name)"
    }
}
```

If we wanted to remove the **return** statements in there, we could *not* write this:

```swift
func greet(name: String) -> String {
    if name == "Taylor Swift" {
        "Oh wow!"
    } else {
        "Hello, \(name)"
    }
}
```

That isn't allowed, because we have actual statements in there – **if** and **else**.
We can use ternary operator instead:

```swift
259 func isTaylor (name : String) -> String{
260     name == "Taylor" ? "Hello Taylor"  : "Hi \(name)!"
261 }
262 print (isTaylor(name: "Hanah"))
263 print (isTaylor(name: "Taylor"))
```

(2 times)

"Hi Hanah!\n"
"Hello Taylor\n"

Parameter Labels

- Using 1 name for parameter
- Using 2 names for parameter : external + internal names

```
266 //Parameter labels : external name (optional) + internal name
267
268 func call ( person: String) //1 parameter name
269 {
270     print("Calling to " + person)
271 }
272
273 func talk (with person: String) //2 parameter names
274 {
275     print("Talking with " + person)
276 }
277
278 call(person: "John") //call func with internal parameter's name
279 talk(with: "Narsh") //call func with external parameter's name
```

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

The parameter is called **to name**, which means externally it's called **to**, but internally it's called **name**.
This gives variables a sensible name inside the function, but means calling the function reads naturally:

```
sayHello(to: "Taylor")
```

- Omitting parameter with underscore

```
282 // Ommitting external name of parameter with '_' (underscore)
283
284 func greet(_ person: String) {
285     print("Hello \(person)")
286 }
287 greet("John")
```

You can now call **greet()** without having to use the **person** parameter name

- Default parameters:

You can give your own parameters a default value just by writing an **=** after its type followed by the default you want to give it.

```swift
//default param

func sum3Num (num1 : Int = 0 , num2 : Int = 0, num3 : Int = 0)
{
    print("Sum = \(num1 + num2 + num3)")
}
sum3Num()                               //=0
sum3Num(num1: 1, num2: 2)               //=3
sum3Num(num1: 4, num2: 5, num3: 6)   //=15
```

✅ Inout parameters:

You can pass in one or more parameters as **inout**, which means they can be changed inside your function, and those changes reflect in the original value outside the function.

(All parameters passed into a Swift function are ***constants***, so you can't change them unless you define it as **inout parameter**)

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

And when calling the function, using an ampersand, **&**, before parameter name

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

✅ Variadic functions:

- Some functions are *variadic* - they accept any number of parameters of the same type.
- For example: **print()** is a variadic function.

- You can make any parameter variadic by writing **...** after its type

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

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

✅ Throwing function:

- We throw errors from functions by marking them as **throws** before their return type, then using the **throw** keyword when something goes wrong.

1. Define an enum that describe the errors we can throw.

These MUST always be based on Swift's existing **Error** type

```
enum PasswordError: Error {
    case obvious
}
```

2. Write a function that throw that error if something goes wrong. Using the **throws** keyword before the function's return value

```
func checkPassword(_ password: String) throws -> Bool {
    if password == "password" {
        throw PasswordError.obvious
    }

    return true
}
```

3. Throwing function: Using do, try, catch

+ do starts a section of code that might cause problems
+ try is used before every function that might throw an error
+ catch lets you handle errors gracefully.

If any errors are thrown inside the **do** block, execution immediately jumps to the **catch** block

```
do {
    try checkPassword("password")
    print("That password is good!")
} catch {
    print("You can't use that password.")
}
```

Example:

```swift
//step 1
enum PasswordErrorHandling : Error {
    case tooSimple
    case tooShort
    case emtyInput
}
//step 2
func checkPassword (_ password: String) throws -> Bool
{
    if (password.isEmpty){
        throw PasswordErrorHandling.emtyInput
    }
    if (password == "password"){
        throw PasswordErrorHandling.tooSimple
    }
    if (password.count < 8){
        throw PasswordErrorHandling.tooShort
    }
    return true

}
```

```swift
//step 3
var myPassword = "lo"

do {
    try checkPassword(myPassword)
    print("You have a good password")
}
catch { error
    print(error) // output: 'tooShort'
    switch error as! PasswordErrorHandling {
    case PasswordErrorHandling.emtyInput:
        print("Empty input")
    case PasswordErrorHandling.tooShort:
        print("Password is too short")
    case PasswordErrorHandling.tooSimple:
        print("Password is too simple")
    }


}
```

REF link: https://www.donnywals.com/working-with-throwing-functions-in-swift/