XI / Optionals

=> Null references – literally when a variable has no value

Handling missing data

=> It allows us to represent the absence of some data – a string that isn't just empty, but literally doesn't exist

=> Can make optionals out of any type. An optional integer might have a number like 0 or 40, but it might have no value at all –but also might be nil – it might not exist.

Exemple :

var age: Int? = nil <- That doesn't hold any number – it holds nothing

=> later learn that age, we can use it:

age = 38

Total score: 6/6 checked

Unwrapping optionals

=> Optional Type might contain something like "Hello" or they might be nil – nothing at all

=> Trying to read a value is unsafe and Swift won't allow it

Exemple 1 :

```
var name: String? = nil


if let unwrapped = name {
    print("\(unwrapped.count) letters")
} else {
    print("Missing name.")
}
```

Exemple 2 :

```
func getUsername() -> String? {
    "Taylor"
}

if let username = getUsername() {
    print("Username is \(username)")
} else {
    print("No username")
}
```

Exemple 3 :

```
let song: String? = "Shake it Off"
if let unwrappedSong = song {
        print("The name has \(unwrappedSong.count) letters.")
}
```

Exemple 4 :

```
let album = "Red"
let albums = ["Reputation", "Red", "1989"]
if let position = albums.firstIndex(of: album) {
        print("Found \(album) at position \(position).")
}
```

NB :

- A common way of unwrapping optionals is with **if let** syntax **which unwraps with a condition.**

- Swift won't let use them without unwrapping them first

Total score: 12/12 checked

Unwrapping with guard

=> Alternative to if let is guard let, which also unwraps optionals. guard let will unwrap an optional for you, but if it finds nil inside it expects you to exit the function, loop, or condition you used it in

=> the major difference between if let and guard let is that your unwrapped optional remains usable after the guard code.

Exemple 1 :

```
func greet(_ name: String?) {
    guard let unwrapped = name else {
        print("You didn't provide a name!")
        return
    }

    print("Hello, \(unwrapped)!")
}
```

Exemple 2 :

```
func double(number: Int?) -> Int? {
        guard let number = number else {
```

```
        return nil
    }

    return number * 2
}

let input = 5
if let doubled = double(number: input) {
    print("\(input) doubled is \(doubled).")      <-  Print 5 doubled is 10
}
```

Exemple 3 :

```
func isLongEnough(_ string: String?) -> Bool {
    guard let string = string else { return false }

    if string.count >= 8 {
        return true
    } else {
        return false
    }
}
if isLongEnough("Mario Odyssey") {
    print("Let's play that!")
}
```

Exemple 4 :

```
func uppercase(string: String?) -> String? {
    guard let string = string else {
        return nil
    }

    return string.uppercased()
}

if let result = uppercase(string: "Hello") {
    print(result)
}
```

Exemple 5 :

```
func describe(occupation: String?) {
	guard let occupation = occupation else {
		print("You don't have a job.")
		return
	}
	print("You are an \(occupation).")
}
let job = "engineer"
describe(occupation: job)
```

Exemple 6 :

```
func league(for skillLevel: Int) -> Int? {
	switch skillLevel {
	case 1:
		fallthrough
	case 2:
		return 3
	case 3:
		return 2
	case 4:
		return 1
	default:
		return nil
	}
}
let allocatedLeague = league(for: 3)!
```

Exemple 7 :

```
struct Dog {
	var name: String
	init?(name: String) {
		guard name == "Lassie" else {
			print("Sorry, wrong dog!")
			return nil
```

```
            }
            self.name = name
        }
}
let dog = Dog(name: "Fido")
```

NB :
Guard requires that we exit the current scope when it's used, which in this case means we must return from the function if it fails. This is not optional: Swift won't compile our code without the return.

Total score: 12/12 checked

Force unwrapping

=> force unwrap the optional: convert from an optional type to a non-optional type.

Exemple 1: => convert value type string to an type **Int**

```
let str = "5"
let num = Int(str)!
```

NB : force unwrap only when you're sure it's safe

Exemple 2 :

```
let url = URL(string: "https://www.apple.com")!     <- safe

let url = URL(string: "https://www.\(website)")!      <- unsafe because string interpolation

 - - - - - - - - - - -

let randomNumber = (1...10).randomElement()!   <- safe
```

```
var items = [Int]()

for i in 1...10 {
    if isLuckyNumber(i) {
        items.append(i)
    }
}

let randomNumber = items.randomElement()!       <- unsafe
```

Total score: 6/6 checked

Implicitly unwrapped optionals

=> unlike regular optionals you don’t need to unwrap them in order to use them: you can use them as if they weren’t optional at all

Exemple :

```
let age: Int! = nil
```

NB :

Because they behave as if they were already unwrapped, you don’t need if let or guard let to use implicitly unwrapped optionals.

However, if you try to use them and they have no value – if they are nil – your code crashes.

Implicitly unwrapped optionals exist because sometimes a variable will start life as nil, but will always have a value before you need to use it.

Because you know they will have a value by the time you need them, it’s helpful not having to write if let all the time

Total score: 6/6 checked

Nil coalescing

=> nil coalescing we can provide a default value
=> use nil coalescing across different types, which isn't allowed.

Exemple 1 :

```
func username(for id: Int) -> String? {
    if id == 1 {
        return "Taylor Swift"
    } else {
        return nil
    }
}
```

-> ID 15 : we'll get back **nil** because the user isn't recognized, but with nil coalescing we can provide a default value of "Anonymous"

- - - - -

```
let user = username(for: 15) ?? "Anonymous"
```

-> That will check the result that comes back from the **username()** function: if it's a string then it will be unwrapped and placed into **user**, but if it has **nil** inside then "Anonymous" will be used instead.

Exemples 2 :

```
var bestPony: String? = "Pinkie Pie"
let selectedPony: String? == bestPony ?? nil

let lightsaberColor: String? = "green"
let color = lightsaberColor ?? "blue"

var captain: String? = "Kathryn Janeway"
let name = captain ?? "Anonymous"

var selectedYear: Int? = nil
let actualYear = selectedYear ?? 1989
```

Total score: 12/12 checked

Optional chaining

=> Swift provides us with a shortcut when using optionals: if you want to access something like a.b.c and b is optional, you can write a question mark after it to enable *optional chaining*: **a.b?.c.**

Exemple 1 :

let names = ["John", "Paul", "George", "Ringo"]

- - - - -
=> We use the **first** property of that array, which will return the first name if there is one or **nil** if the array is empty

let beatle = names.first?.uppercased() <- return Optional("JOHN")

=> if **first** returns **nil** then Swift won't try to uppercase it, and will set **beatle** to **nil** immediately

---

Exemple 2 :

=> list of names and want to find where they should be placed based on the first letter of their surname:

let names = ["Vincent": "van Gogh", "Pablo": "Picasso", "Claude": "Monet"]
let surnameLetter = names["Vincent"]?.first?.uppercased()

let surnameLetter = names["Vincent"]?.first?.uppercased() ?? "?"

Exemple 3 :

```
let songs: [String]? = [String]()
let finalSong = songs?.last?.uppercased()
```

Exemple 4 :

```
func albumReleased(in year: Int) -> String? {
        switch year {
        case 2006: return "Taylor Swift"
        case 2008: return "Fearless"
        case 2010: return "Speak Now"
        case 2012: return "Red"
        case 2014: return "1989"
        case 2017: return "Reputation"
        default: return nil
        }
}
let album = albumReleased(in: 2006)?.uppercased()
```

Exemple 5 :

```
let shoppingList = ["eggs", "tomatoes", "grapes"]
let firstItem = shoppingList.first?.appending(" are on my shopping list")
```

More details : https://andybargh.com/optional-chaining/

Total score: 12/12 checked

Optional try

=> using try you must catch all errors that can be thrown.

Exemple 1 :

```
enum PasswordError: Error {
    case obvious
}

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

    return true
}


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

- - - — -

=> That runs a throwing function, using **do**, **try**, and **catch** to handle errors

-> The first is **try?**, and changes throwing functions into functions that return an optional. If the function throws an error you'll be sent **nil** as the result, otherwise you'll get the return value wrapped as an optional.

Using Try :

```
if let result = try? checkPassword("password") {
    print("Result was \(result)")
} else {
    print("D'oh.")
}
```

- - - — - -

-> The other alternative is **try!**, which you can use when you know for sure that the function will not fail. If the function *does* throw an error, your code will crash:

```
try! checkPassword("sekrit")
print("OK!")
```

_____

Exemple 2 : (alternative)

```
do {
    let result = try runRiskyFunction()
    print(result)
} catch {
    // it failed!
}
```

- - - - - - - - -

-> to use **try?** to convert a throwing function call into an optional. If the function succeeds, its return value will be an optional containing whatever you would normally have received back, and if it fails the return value will be an optional set to nil.

=> You can combine **try?** with **if let**

```
if let result = try? runRiskyFunction() {
    print(result)
}
```

NB :

use optional try a heck of a lot in my own code, because it does wonders for letting me focus on the problem at hand.

Total score: 6/6 checked

_____
_____

Failable initializers

=> When talking about force unwrapping, I used this code:

Exemple 1 :

```
let str = "5"
let num = Int(str)
```

NB:
That converts a string to an integer, but because you might try to pass any string there what you actually get back is an *optional* integer.

This is a *failable initializer*: an initializer that might work or might not. You can write these in your own structs and classes by using **init?()** rather than **init()**, and return **nil** if something goes wrong.

The return value will then be an optional of your type, for you to unwrap however you want.

Exemple 2 :

-> We could code a **Person** struct that must be created using a nine-letter ID string. If anything other than a nine-letter string is used we'll return **nil**, otherwise we'll continue as normal.

```
struct Person {
    var id: String

    init?(id: String) {
        if id.count == 9 {
            self.id = id
        } else {
            return nil
        }
    }
}
```

NB :

Making a failable initializer takes two steps:

1. Write your initializer as **init?()** rather than **init()**

2. Return nil for any paths that should fail

You can have as many failing paths as you need, but you don't need to worry about the success path – if you don't return nil from the method, Swift assumes you mean everything worked correctly.

Exemple : -> here's an **Employee** struct that has a failable initializer with two checks:

```
struct Employee {
    var username: String
    var password: String

    init?(username: String, password: String) {
        guard password.count >= 8 else { return nil }
        guard password.lowercased() != "password" else { return nil }

        self.username = username
        self.password = password
    }
}
```

- - - - - - - - -

```
let tim = Employee(username: "TimC", password: "app1e")
let craig = Employee(username: "CraigF", password: "ha1rf0rce0ne")
```

NB :

The first of those will be an optional set to nil because the password is too short, but the second will be an optional set to a valid **User** instance.

Failable initializers give us the opportunity to back out of an object's creation if validation checks fail.

In the previous case that was a password that was too short, but you could also check whether the username was taken already, whether the password was the same as the username, and so on.

_____

Other exemples :

```
struct Password {
	var text: String
	init?(input: String) {
		if input.count < 6 {
			print("Password too short.")
			return nil
		}
		text = input
	}
}
let password = Password(input: "hell0")
```

Total score: 12/12 checked

_____

Typecasting

=> Swift must always know the type of each of your variables, but sometimes you know more information than Swift does.

Example:

```
class Animal { }
class Fish: Animal { }

class Dog: Animal {
   func makeNoise() {
      print("Woof!")
   }
}
```

- - - - - - -

-> We can create a couple of fish and a couple of dogs, and put them into an array

```
let pets = [Fish(), Dog(), Fish(), Dog()]
```

- - - - - - -

-> If we want to loop over the **pets** array and ask all the dogs to bark, we need to perform a typecast: Swift will check to see whether each pet is a **Dog** object, and if it is we can then call **makeNoise()**.

-> uses a new keyword called **as?**, which returns an optional: it will be **nil** if the typecast failed, or a converted type otherwise

```
for pet in pets {
    if let dog = pet as? Dog {
        dog.makeNoise()
    }
}
```

Typical exemple to read as a proper situation TypeCast :

-> class hierarchy:

```
class Person {
    var name = "Anonymous"
}

class Customer: Person {
    var id = 12345
}

class Employee: Person {
    var salary = 50_000
}
```

- - - -

-> I've used default values for each property so we don't need to write an initializer.
We can create an instance of each of those, and add them to the same array:

```
let customer = Customer()
let employee = Employee()
let people = [customer, employee]
```

- - - - -

-> Because both **Customer** and **Employee** inherit from **Person**, Swift will consider that **people** constant to be a **Person** array. So, if we loop over **people** we'll only be able to access the **name** of each item in the array – or at least we *would* only be able to do that, if it weren't for type casting:

```
for person in people {
    if let customer = person as? Customer {
        print("I'm a customer, with id \(customer.id)")

    } else if let employee = person as? Employee {
        print("I'm an employee, earning $\(employee.salary)")
    }
}
```

Exemple :

```
class Transport { }
class Train: Transport {
        var type = "public"
}
class Car: Transport {
        var type = "private"
}
let travelPlans = [Train(), Car(), Train()]
for plan in travelPlans {
        if let train = plan as? Train {
                print("We're taking \(train.type) transport.")
        }
}
```

Exemple :

```
class Bird {
        var wingspan: Double? = nil
}
class Eagle: Bird { }
```

```
let bird = Eagle()
if let eagle = bird as? Eagle {
        if let wingspan = eagle.wingspan {
                print("The wingspan is \(wingspan).")
        } else {
                print("This bird has an unknown wingspan.")
        }
}
```

Total score: 12/12 checked

Optionals summary

1. Optionals let us represent the absence of a value in a clear and unambiguous way.

2. Swift won't let us use optionals without unwrapping them, either using **if let** or using **guard let**.

3. You can force unwrap optionals with an exclamation mark, but if you try to force unwrap **nil** your code will crash.

4. Implicitly unwrapped optionals don't have the safety checks of regular optionals.

5. You can use nil coalescing to unwrap an optional and provide a default value if there was nothing inside.

6. Optional chaining lets us write code to manipulate an optional, but if the optional turns out to be empty the code is ignored.

7. You can use **try?** to convert a throwing function into an optional return value, or **try!** to crash if an error is thrown.

8. If you need your initializer to fail when it's given bad input, use **init?()** to make a failable initializer.

9. You can use typecasting to convert one type of object to another.