II / Complex data types : Suite

---

6 / Enumerations

Way of defining groups of related values in a way that makes them easier to use

=> to represent the success or failure of some work you were doing

Bad Exemple :

```
let result = "failure"
let result2 = "failed"
let result3 = "fail"
```

=> Enums we can define a Result type that can be either success or failure

Good Exemple :

```
enum Result {
   case success
   case failure
}


let result4 = Result.failure
```

=> This stops you from accidentally using different strings each time.

Total score: 6/6 checked

---

7 / Enum associated values

=> Define an enum that stores various kinds of activities:

Exemple :

```
enum Activity {
    case bored
    case running
    case talking
    case singing
}
```

=> Define someone is talking, but we don’t know what they are talking *about*, or we can know that someone is running, but we don’t know where they are running *to* :

Exemple :

```
enum Activity {
    case bored
    case running(destination: String)
    case talking(topic: String)
    case singing(volume: Int)
}
```

=> Be more precise – we can say that someone is talking about football:

Exemple :

```
let talking = Activity.talking(topic: "football")
```

=> **we might create a Weather enum with three cases:**

```
enum Weather {
    case sunny
    case windy(speed: Int)
    case rainy(chance: Int, amount: Int)
}
```

Total score: 6/6 checked

---

8 / Enum raw values

Need to be able to assign values to enums so they have meaning

=> Create a **Planet** enum that stores integer values for each of its cases

Exemple :

```
enum Planet: Int {
    case mercury
    case venus
    case earth
    case mars
}

let earth = Planet(rawValue: 2)
```

=> Assign one or more cases a specific value, and Swift will generate the rest

Exemple :

```
enum Planet: Int {
    case mercury = 1
    case venus
    case earth
    case mars
}

print(Planet.mercury.rawValue) // 1
print(Planet.venus.rawValue) // 2
print(Planet.earth.rawValue) // 3
print(Planet.mars.rawValue) // 4
```

NB : Heart value is now 3

=> can be handy if you’re using them, for example, with tracking events:

Exemple :

```
enum TrackingEvent: String {
    case loggedIn = "logged_in"
}
```

=> **Iterating over all enum cases**

Exemple :

```
enum SocialPlatform: String, CaseIterable {
    case twitter
    case facebook
    case instagram
}

print(SocialPlatform.allCases) // twitter, facebook, instagram
```

=> Enums and Equatable

Exemple :

```
enum SocialPlatform {
    case twitter
    case facebook
    case instagram
}

let mostUsedPlatform = SocialPlatform.twitter

if mostUsedPlatform == .facebook {
    print("Fake news")
} else {
    print("You're totally right!")
}
```

Total score: 6/6 checked

Review

=> Lets create simple enum as multiple case:
Exemple 1:

```
enum WeatherType {
	case sun
	case cloud
	case rain
	case wind
	case snow
}

func getHatersStatus(weather: WeatherType) -> String ? {
	switch weather {
	case .sun:
		return nil
	case .cloud  , case  .wind, case .snow:
		return ''dislike''
	case .rain :
		return ''Hate''
	}
}

getHaterStatus(weather: .cloud)
```

_____

Exemple 2:

=> enum can have value

```
enum WeatherType {

	case sun
	case cloud
	case rain
	case wind (speed:Int)
	case snow
}
```

```
func getHaterStatus(weather:WheatherType) -> String? {

	switch weather {
	case .sun:
		return nil
	case .wind(let speed) where speed < 10:		<- use let to get the
value inside the enum
		return ''houha''
	case .cloud, .wind:
		return ''dislike''
	case .rain, . snow:
		return ''hate''
	}
}
```