VIII / Structs : Part one

Creating structs

=> Struct stores its name as a string(for example). Variables inside structs are called *properties*, **so this is a struct with its propertys:**

Exemple :

```
struct Sport {
   var name: String
}
```

- - - - - - - - - - - - -

=> use an instance of it:

```
var tennis = Sport(name: "Tennis")
print(tennis.name)
```

– - - - - - - - - - - - -

=> It can change them just like regular variables:

```
tennis.name = "Lawn tennis"
```

NB: Use structs when you have some fixed data you want to send or receive multiple times.

Exemple :

Use This =>

```
struct User {
   var name: String
   var age: Int
   var city: String
}
```

Instead of this =>

```
func authenticate(_ user: (name: String, age: Int, city: String)) { ... }
func showProfile(for user: (name: String, age: Int, city: String)) { ... }
func signOut(_ user: (name: String, age: Int, city: String)) { ... }
```

Total score: 12/12 checked

Computed properties

=> Swift has a different kind of property called a *computed* property – a property that runs code to figure out its value
=> Computed properties must always have an explicit type
=> Constants cannot be computed properties.

Exemple 1 : Stored property to the struct, then a computed property

```
struct Sport {
    var name: String
    var isOlympicSport: Bool  <- Stored property

    var olympicStatus: String {  <- computed property
        if isOlympicSport {
            return "\(name) is an Olympic sport"
        } else {
            return "\(name) is not an Olympic sport"
        }
    }
}
```

- - - - - - - - - -

=> Computed properties returns different values depending on the other properties

```
let chessBoxing = Sport(name: "Chessboxing", isOlympicSport: false)
print(chessBoxing.olympicStatus)
```

Exemple 2 :

```
struct Wine {
	var age: Int
	var isVintage: Bool
	var price: Int {
		if isVintage {
			return age + 20
		} else {
			return age + 5
		}
	}
}
let malbec = Wine(age: 2, isVintage: true)
print(malbec.price)
```

```
struct Medicine {
	var amount: Int
	var frequency: Int
	var dosage: String {
		return "Take \(amount) pills \(frequency) times a day."
	}
}
```

Exemple 3 :

```
struct Dog {
	var breed: String
	var cuteness: Int
	var rating: String {
		if cuteness < 3 {
			return "That's a cute dog!"

		} else if cuteness < 7 {
			return "That's a really cute dog!"

		} else {
			return "That a super cute dog!"
		}
```

```
    }
}
let luna = Dog(breed: "Samoyed", cuteness: 11)
print(luna.rating)
```

NB : Two variations: stored properties :

- Stored properties where value is stashed away in some memory to be used later
  - Stored property will be much faster than using a computed property
- Computed Properties where a value is recomputed every time it's called
  - Saves you from having to calculate its value and store it somewhere.

Total score: 12/12 checked

Property observers

=> observers let run code before or after any property changes

Exemple : struct that tracks a task and a completion percentage

```
struct Progress {
   var task: String
   var amount: Int
}
```

- - - - - - - - - - -

=> instance of that struct and adjust its progress over time

```
var progress = Progress(task: "Loading data", amount: 0)
progress.amount = 30
progress.amount = 80
progress.amount = 100
```

- - - - - - - - - - -

=> print a message every time **amount** changes, use a **didSet** property observer for that. This will run some code every time **amount** changes:

```
struct Progress {
   var task: String
   var amount: Int {

      didSet {. <- property observer
         print("\(task) is now \(amount)% complete")
      }
   }
}
```

NB : **willSet** and **didSet** let us attach observers to properties will run some code when those change :

- Before property change -> use willSet
- After property change -> use didset

Total score: 12/12 checked

Methods (function which belong to a struct)

=> those functions can use the properties of the struct as they need to

Exemple 1 :

```
struct City {
   var population: Int

   func collectTaxes() -> Int { <- method
      return population * 1000
   }
}
```

—- - - — - - - - -

=> That method belongs to the struct, so we call it on instances of the struct

```
let london = City(population: 9_000_000)

london.collectTaxes()
```

Exemple 2 :

```
struct Venue {
	var name: String
	var maximumCapacity: Int
	func makeBooking(for people: Int) {
		if people > maximumCapacity {
			print("Sorry, we can only accommodate \
(maximumCapacity).")
		} else {
			print("\(name) is all yours!")
		}
	}
}
```

Total score: 12/12 checked

Mutating methods

=> If a struct has a variable property but the instance of the struct was created as a constant, that property can't be changed :
– Struct is constant, so all its properties are also constant regardless of how they were created.
_ It won't let you write methods that change properties unless you specifically request it

Exemple 1 : To change a property inside a method => using the **mutating** keyword

```
struct Person {
   var name: String
```

```
    mutating func makeAnonymous() {
        name = "Anonymous"
    }
}
```

— - - - - - - - - -

=> Swift will only allow that method to be called on Person instances that are variables:

```
var person = Person(name: "Ed")
person.makeAnonymous()
```

———————————————

Exemple 2 :

```
struct Book {
        var totalPages: Int
        var pagesLeftToRead = 0
        mutating func read(pages: Int) {
                if pages < pagesLeftToRead {
                        pagesLeftToRead -= pages
                } else {
                        pagesLeftToRead = 0
                        print("I'm done!")
                }
        }
}
```

Exemple 3 :

```
struct Bicycle {
        var currentGear: Int
        mutating func changeGear(to newGear: Int) {
                currentGear = newGear
                print("I'm now in gear \(currentGear).")
        }
}
```

NB : A method that is *not* marked as mutating cannot call a mutating function – you must mark them both as mutating.

Total score: 12/12 checked

Properties and methods of strings

=> It turn out that strings are structs
=> they have their own methods and properties we can use to query and manipulate the string.

Exemple 1 :

```
let string = "Do or do not, there is no try."


print(string.count) <- the number of characters in a string

print(string.hasPrefix("Do")) <- method that returns true if the string starts with specific letters

print(string.uppercased()) <- return a uppercase a string

print(string.sorted()) <- sort the letters of the string into an array

 print(string.isEmpty) <- check if there is any letters rather than -> print(string.count == 0
```

Exemple 2 :

```
let quote = "Time is an illusion. Lunchtime doubly so."
quote.contains("Lunch")
```

NB : Almost all of Swift’s core types are implemented as structs, including strings, integers, arrays, dictionaries, and even Booleans

Total score: 6/6 checked

Properties and methods of arrays

=> Arrays are structs, they have their own methods and properties

Exemple 1 :

```
var toys = ["Woody"]

print(toys.count) <- return number of items in an array

toys.append("Buzz") <- add a new item

toys.firstIndex(of: "Buzz") <- locate any item inside an array

print(toys.sorted()) <- sort the items of the array alphabetically

toys.remove(at: 0) <- remove an item
```

Exemple 2 :

```
var usedNumbers = [Int]()
for i in 1...10 {
	usedNumbers.append(i)
}
usedNumbers.count > 5
```

Total score: 12/12 checked