

VIII / Structs : Part two

Initializers

=> By default, all Swift structs get a synthesized memberwise initializer by default

- Which mean initializer that accepts values for each of the struct's properties
- Initializers cannot finish until all properties have a value

Exemple 1 :

```
struct Employee {  
    var name: String  
    var yearsActive = 0  
}
```

```
let roslin = Employee(name: "Laura Roslin")  
let adama = Employee(name: "William Adama", yearsActive: 45)
```

- - - - -

=> custom initializer that created anonymous employees

```
struct Employee {  
    var name: String  
    var yearsActive = 0  
  
    init() {    <- custom initializer  
        self.name = "Anonymous"  
        print("Creating an anonymous employee...")  
    }  
}
```

!! It no longer rely on the memberwise initializer, so the following exemple would no longer be allowed !! :

```
let roslin = Employee(name: "Laura Roslin")
```

=> as soon as you add a custom initializer for your struct, the default memberwise initializer goes away

=> If it need to stay, move custom initializer to an extension -> init()

```
struct Employee {  
    var name: String  
    var yearsActive = 0  
}
```

```
extension Employee {  
    init() {                <- extension for custom initializer  
        self.name = "Anonymous"  
        print("Creating an anonymous employee...")  
    }  
}
```

```
// creating a named employee now works  
let roslin = Employee(name: "Laura Roslin")  
  
// as does creating an anonymous employee  
let anon = Employee()
```

Exemple 2 :

```
struct Country {  
    var name: String  
    var usesImperialMeasurements: Bool  
    init(countryName: String) {  
        name = countryName  
        let imperialCountries = ["Liberia", "Myanmar", "USA"]  
        if imperialCountries.contains(name) {
```

```

        usesImperialMeasurements = true
    } else {
        usesImperialMeasurements = false
    }
}
}

```

Exemple 3 :

```

struct Cabinet {
    var height: Double
    var width: Double
    var area: Double
    init (itemHeight: Double, itemWidth: Double) {
        height = itemHeight
        width = itemWidth
        area = height * width
    }
}
let drawers = Cabinet(itemHeight: 1.4, itemWidth: 1.0)

```

Total score: 12/12 Checked

Referring to the current instance

=> Inside methods you get a special constant called -> **self**, which points to whatever instance of the struct is currently being used

- **self** helps to distinguish between the property and the parameter – **self.propertyName** refers to the property, whereas **name** refers to the parameter from **init()** initializer.

Exemple 1 :

```

struct Person {
    var name: String

```

```
init(name: String) {  
    print("\(name) was born!")  
    self.name = name  
}  
}
```

Exemple 2 :

```
struct Student {  
    var name: String  
    var bestFriend: String  
  
    init(name: String, bestFriend: String) {  
        print("Enrolling \(name) in class...")  
        self.name = name  
        self.bestFriend = bestFriend  
    }  
}
```

Exemple 3 :

```
struct Conference {  
    var name: String  
    var location: String  
  
    init(name: String, location: String) {  
        self.name = name  
        self.location = location  
    }  
}  
  
let wwdc = Conference(name: "WWDC", location: "San Jose")
```

Exemple 4 :

```
struct Language {  
    var nameEnglish: String  
    var nameLocal: String
```

```

var speakerCount: Int

init(english: String, local: String, speakerCount: Int) {
    self.nameEnglish = english
    self.nameLocal = local
    self.speakerCount = speakerCount
}
}

let french = Language(english: "French", local: "français", speakerCount:
220_000_000)

```

Exemple 5 :

```

struct Character {
    var name: String
    var actor: String
    var probablyGoingToDie: Bool

    init(name: String, actor: String) {
        self.name = name
        self.actor = actor
        if self.actor == "Sean Bean" {
            probablyGoingToDie = true
        } else {
            probablyGoingToDie = false
        }
    }
}

```

NB : Common reason for using self is inside an initializer, where we want parameter names that match the property names type

Total score: 12/12 checked

Lazy properties

=> As a performance optimization, Swift lets create some properties

only when they are needed

=> lazy properties let delay the creation of a property until it's actually used

=> A lazy property can be an instance of a different struct

Exemple :

```
struct FamilyTree {      <- this FamilyTree struct – it doesn't do much
  init() {
    print("Creating family tree!")
  }
}
```

- - - - -

=> If we need struct as a property inside a Person struct, like this:

```
struct Person {
  var name: String
  var familyTree = FamilyTree()

  init(name: String) {
    self.name = name
  }
}
```

```
var ed = Person(name: "Ed")
```

- - - - -

=> if we didn't always need the family tree for a particular person but at least once (it's first accessed) :

```
struct Person {
  var name: String
  Lazy var familyTree = FamilyTree()  <- lazy property

  init(name: String) {
    self.name = name
  }
}
```

```
var ed = Person(name: "Ed")
```

```
ed.familyTree
```

NB : Reasons why store computed properties over a lazy property :

1. Using lazy properties can accidentally produce work where you don't expect it. For example, if you're building a game and access a complex lazy property for the first time it might cause your game to slow down, so it's much better to do slow work up front and get it out of the way.
2. Lazy properties always store their result, which might either be unnecessary (because you aren't use it again) or be pointless (because it needs to be recalculated frequently).
3. Because lazy properties change the underlying object they are attached to, you can't use them on constant structs.

Total score: 6/6 checked

Static properties and methods (scope notion -> access to each instances of struct)

=> All the properties and methods belonged to individual instances of structs

=> It's possible to ask Swift to share specific properties and methods across all instances of the struct by declaring them as -> *static*

=> Referencing a static property inside a regular method isn't allowed -> the right way :

-> structName.property

Exemple :

```
struct Student {  
    static var classSize = 0  
    var name: String  
  
    init(name: String) {  
        self.name = name  
        Student.classSize += 1  
    }  
}
```

```
}  
}
```

=> Because the **classSize** property belongs to the struct itself rather than instances of the struct
We need to read it like that :

```
print(Student.classSize)
```

Exemple 2 :

```
struct Unwrap {  
    static let appURL = "https://itunes.apple.com/app/id1440611372"  
}
```

=> Without the **static** keyword I'd need to make a new instance of the **Unwrap** struct just to read the fixed app URL

=> We can use both a static property and a static method to store some random entropy in the same struct

Exemple 3 :

```
static var entropy = Int.random(in: 1...1000)  
  
static func getEntropy() -> Int {  
    entropy += 1  
    return entropy  
}
```

Exemple 4 :

```
struct NewsStory {  
    static var breakingNewsCount = 0
```



```

static var regularNewsCount = 0
var headline: String

init(headline: String, isBreaking: Bool) {
    self.headline = headline
    if isBreaking {
        NewsStory.breakingNewsCount += 1
    } else {
        NewsStory.regularNewsCount += 1
    }
}

```

Exemple 5 :

```

struct Pokemon {
    static var numberCaught = 0
    var name: String

    static func catchPokemon() {
        print("Caught!")
        Pokemon.numberCaught += 1
    }
}

```

NB : The reason to use static is to store common functionality you use across an entire app

Total score: 12/12 checked

Access control

1. Private

=> lets you restrict which code can use properties and methods

=> Make their property be private so it can't read it from outside the struct

=> access control keywords it' a restriction about how different parts of code can be accessed

=> As private property, Swift is unable to generate its memberwise initializer

Exemple 1 :

```
struct Person {  
    private var id: String  
  
    init(id: String) {  
        self.id = id  
    }  
}
```

- - - - -

=> This way only methods inside Person can read the id property :

```
struct Person {  
    private var id: String  
  
    init(id: String) {  
        self.id = id  
    }  
  
    func identify() -> String {  
        return "My social security number is \(id)"  
    }  
}
```

Exemple 2 :

```
struct Person {  
    private var socialSecurityNumber: String  
  
    init(ssn: String) {  
        socialSecurityNumber = ssn  
    }  
}
```

```
}
```

```
let sarah = Person(ssn: "555-55-5555")
```

Exemple 3 :

```
struct Office {  
    private var passCode: String  
    var address: String  
    var employees: [String]  
  
    init(address: String, employees: [String]) {  
        self.address = address  
        self.employees = employees  
        self.passCode = "SEKRIT"  
    }  
}
```

```
let monmouthStreet = Office(address: "30 Monmouth St", employees:  
["Paul Hudson"])
```

2.Public

=> Another common option is public, which lets all other code use the property or method

Ref :<https://docs.swift.org/swift-book/LanguageGuide/AccessControl.html>

Total score: 12/12 checked

Summarize

1. You can create your own types using structures, which can have their own properties and methods.
2. You can use stored properties or use computed properties to calculate values on the fly.

3. If you want to change a property inside a method, you must mark it as **mutating**.
4. Initializers are special methods that create structs. You get a memberwise initializer by default, but if you create your own you must give all properties a value.
5. Use the **self** constant to refer to the current instance of a struct inside a method.
6. The **lazy** keyword tells Swift to create properties only when they are first used.
7. You can share properties and methods across all instances of a struct using the **static** keyword.
8. Access control lets you restrict what code can use properties and methods.