IX : Classes

Creating a classe

=> similar to structs , create new data types with properties and methods
However, they have five important differences :

- A) Classes never come with a memberwise initializer
- B) One class can be built upon (“inherit from”) another class (subclass), gaining its properties and methods.
- C) Copies of structs are always unique, whereas copies of classes actually point to the same shared data.
- D) Classes have deinitializers, which are methods that are called when an instance of the class is destroyed, but structs do not.
- E) Variable properties in constant classes can be modified freely, but variable properties in constant structs cannot.

Simple Exemple :

```
class Dog {
    var name: String
    var breed: String

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

=> instance look the same than struct

let poppy = Dog(name: "Poppy", breed: "Poodle")

Total score: 12/12 checked

B) Class inheritance

=> class can create subclass based on an existing class (parent's class)
=> it inherits all the properties and methods of the original class, and can add its own on top

Exemple :

```
class Dog {
   var name: String
   var breed: String

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

- - - - - - - -

=> create a new class based on that one called **Poodle**. It will inherit the same properties and initializer as **Dog** by default:

```
class Poodle: Dog {
   init(name: String) {
      super.init(name: name, breed: "Poodle") <- Super.init() allow
initilization from parent's inheritance
                  property to create  its own  characteristics inherited
                                                      from its parent's class
   }
}
```

Exemple 2 :

```
class Instrument {
	var name: String
	init(name: String) {
		self.name = name
	}
}
class Piano: Instrument {
	var isElectric: Bool

	init(isElectric: Bool) {
		self.isElectric = isElectric
		super.init(name: "Piano")
	}
}
```

Total score: 12/12 checked

Overriding methods

=> Child classes can replace parent methods with their own implementations

Exemple :

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

- - - - - - - -

=> If we create a new Poodle class that inherits from Dog, it will inherit the makeNoise() method

```
class Poodle: Dog {
}

let poppy = Poodle()
poppy.makeNoise()
```

- - - - — - - - -

=> Method overriding allows us to change the implementation of makeNoise() for the Poodle class

```
class Poodle: Dog {
    override func makeNoise() {
        print("Yip!")
    }
}
```

poppy.makeNoise() -> print “Yip!”

NB : Swift requires us to use override func rather than just func when overriding a method – it stops you from overriding a method by accident

NB :
Swift makes us use the **override** keyword before overriding functions :

- If you use it when it isn’t needed (because the parent class doesn’t declare the same method) then you’ll get an error. This stops you from mistyping things, such as parameter names or types, and also stops your override from failing if the parent class changes its method and you don’t follow suit.

- If you *don’t* use it when it *is* needed, then you’ll also get an error. This stops you from accidentally changing behavior from the parent class.

Note totale : 12/12 checked

Final classes

=> sometimes it’s usefull to disallow other developers from building other class from the parent’s class

=> us a final keyword : declare a class as being final, no other class can inherit from it

=> This way we have to use the class the way it was written

Exemple :

```
final class Dog {
   var name: String
   var breed: String

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

NB : many people instinctively declare their classes as final to mean

Total score: 12/12 checked

C) Copying objects

=> !! The third difference between classes and structs is how they are copied. When you copy a struct, both the original and the copy are different things – changing one won’t change the other. When you copy a *class*, both the original and the copy point to the *same* thing, so changing one *does* change the other.!!

Exemple :

```
class Singer {
   var name = "Taylor Swift"
```

}

- - - - - - - - -

=> an instance of that class:

```
var singer = Singer()
print(singer.name)
```

- - - - - - - - - -

=> create a second variable from the first one and change its name:

```
var singerCopy = singer
singerCopy.name = "Justin Bieber"
```

- - - - - - - - - - -

=> **singer** and **singerCopy** point to the same object in memory, so when we print the singer name again we'll see "Justin Bieber"

```
print(singer.name)
```

NB : Compare to struct if Singer were a struct then we would get "Taylor Swift" printed a second time

=> structs always have their own unique data, and changing a copy does not affect the others

=> In this case, using a class rather than a struct sends a strong message that we want the data to be shared somehow, rather than having lots of distinct copies.

Class copy way :

Exemple 1:

```
class Starship {
    var maxWarp = 9.0
}
```

```
var voyager = Starship()
voyager.maxWarp = 9.975

var enterprise = voyager
enterprise.maxWarp = 9.6

print(voyager.maxWarp)
print(enterprise.maxWarp)
```

_____

Exemple 2 :

```
class Author {
	var name = "Anonymous"
}
var dickens = Author()
dickens.name = "Charles Dickens"

var austen = dickens
austen.name = "Jane Austen"

print(dickens.name)
print(austen.name)
```

_____

Exemple 3 :

```
class Hater {
	var isHating = true
}
var hater1 = Hater()
var hater2 = hater1

hater1.isHating = false

print(hater1.isHating)
print(hater2.isHating)
```

_____

Exemple 4 :

```
class Hospital {
        var onCallStaff = [String]()
}
var londonCentral = Hospital()
var londonWest = londonCentral

londonCentral.onCallStaff.append("Dr Harlan")
londonWest.onCallStaff.append("Dr Haskins")

print(londonCentral.onCallStaff.count)
print(londonWest.onCallStaff.count)
```

Total score: 12/12 checked

D) Deinitializers

=> code that gets run when an instance of a class is destroyed

Exemple :

```
class Person {
    var name = "John Doe"

    init() {
        print("\(name) is alive!")
    }

    func printGreeting() {
        print("Hello, I'm \(name)")
    }
}
 - - - - - - - - -
```

=> create a few instances of the Person class inside a loop then each time the loop goes around a new person will be created then destroyed:

```
for _ in 1...3 {
    let person = Person()
```

```
    person.printGreeting()
}
```

- - - - - - - - -

=> deinitializer will be called when the Person instance is being destroyed:

```
deinit {
    print("\(name) is no more!")
}
```

NB : Classes should read like chapters. So the deinitializer goes at the end, it's the ~fin~ of the class!”
=> The job of deinitializers is to tell us when a class instance was destroyed

Total score: 12/12 checked

E) Mutability

=> The final difference between classes and structs is the way they deal with constants. If you have a constant struct with a variable property, that property can’t be changed because the struct itself is constant

=> This difference means you can change any variable property on a class even when the class is created as a constant

Exemple :

```
class Singer {
    var name = "Taylor Swift"
}

let taylor = Singer()
taylor.name = "Ed Sheeran"
print(taylor.name)
```

- - - - - - - - — - - -

=> to stop that from happening you need to make the property constant

```
class Singer {
    let name = "Taylor Swift"
}
```

NB :

- Variable classes can have variable properties changed
- Constant classes can have variable properties changed
- Variable structs can have variable properties changed
- Constant structs *cannot* have variable properties changed

Total score: 12/12 checked