SWIFT INTRO : CLASSES

✅ Classes

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

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

Creating instances of that class looks just the same as if it were a struct

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

Choose Option

Class	Struct
classes never come with a memberwise initializer. If you have properties in your class, you MUST always create your own initializer. **They need initializers if you don't give their properties default values.**	Have memberwise initializer

(Inheritance trait) One class can be built upon (“inherit from”) another class, gaining its properties and methods.	
copies of classes point to the same shared data. So changing one *does* change the other.	When you copy a struct, both the original and the copy are different things – changing one won’t change the other. They are unique
have **deinitializers**, which are methods that are called when an instance of the class is destroyed	No deinitializer
Variable properties in constant classes can be modified freely. Class don't need **mutating** keyword ahead of function	variable properties in constant structs cannot.
--> Variable classes can have variable properties changed	--> Variable structs can have variable properties changed
--> Constant classes can have variable properties changed	--> Constant structs **CANNOT** have variable properties changed

- Class inheritance
  - you can create a class based on an existing class – it inherits all the properties and methods of the original class, and can add its own on top
  - the class you inherit from is called the “parent” or “super” class, and the new class is called the “child” class.
  - Swift only support inheriting from only 1 class

Ex:
Parent class

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

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

Child class

```
class Poodle: Dog {
    init(name: String) {
        super.init(name: name, breed: "Poodle")
    }
}
```

NOTE: Swift always makes you call **super.init() from child classes – just in case the parent class does some important work when it's created.

🟡 Overriding method

- Child classes can replace parent methods with their own implementations – a process known as ***overriding***.
- Swift requires us to use **override func** rather than just **func** when overriding a method

EX:
Dog class have **makeNoise()** function

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

If we create a new **Poodle** class that inherits from **Dog**, it will inherit the **makeNoise()** method. So, this will print “Woof!”:

```
class Poodle: Dog {
}

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

When override makeNoise() in Poodle(), must use **override func** :

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

Therefore , **poppy.makeNoise()** will print “Yip!” rather than “Woof!”.

- Final classes
  - When you declare a class as being final, no other class can inherit from it. Meaning you can’t override your methods in order to change your behavior – they need to use your class the way it was written.

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

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

🟡 Copying objects

When you copy a *class*, both the original and the copy point to the *same* thing, so changing one *does* change the other.

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

Because of the way classes work, both **singer** and **singerCopy** point to the same object in memory => **singer.name** will also be changed to "Justin Bieber" no matter what value it hold before.

🟡 Deinitializer

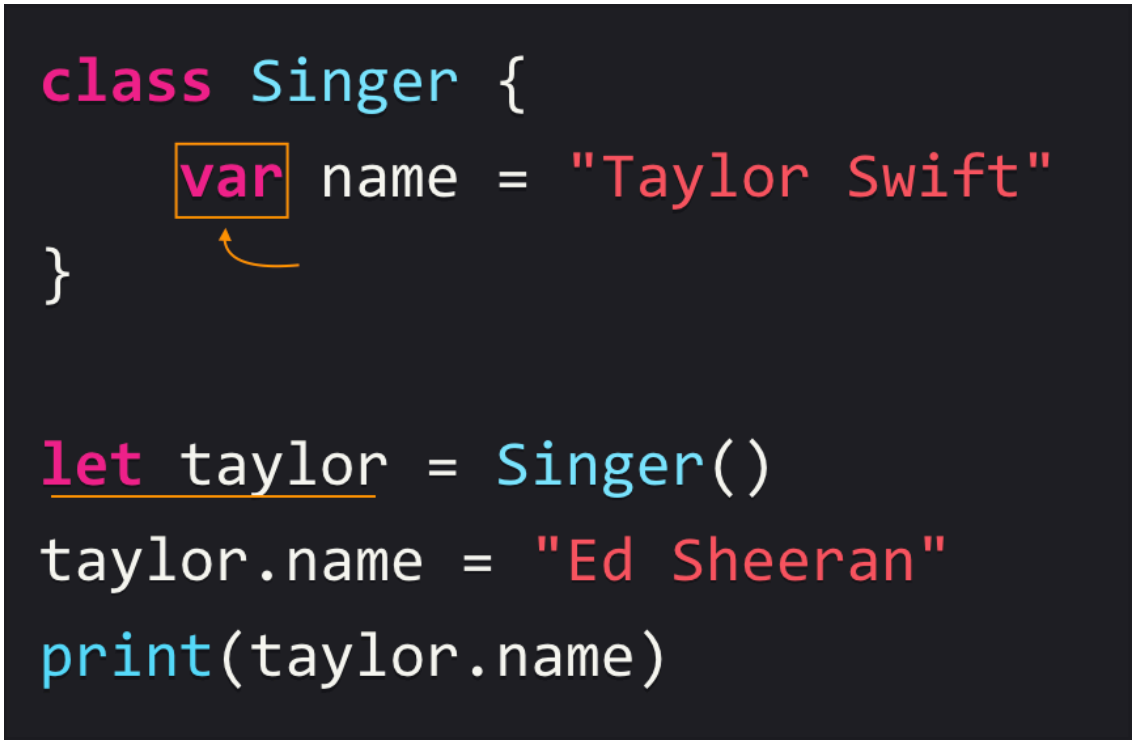
- *deinitializers* – code that gets run when an instance of a class is destroyed.
- The job of deinitializers is to tell us when a class instance was destroyed.
- Using **deinit** keyword

Struct	Class

the struct is destroyed when whatever owns it no longer exists.	Classes have complex copying behavior that means several copies of the class can exist in various parts of your program. All the copies point to the same underlying data
So, if we create a struct inside a method and the methods ends, the struct is destroyed.	So it's harder to tell when the actual class instance is destroyed – when the final variable pointing to it has gone away.

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

- Mutability
  - If you have a **constant *class*** with a **variable property**, that property *can* be changed.
  - Classes don't need the **mutating** keyword with methods that change properties; that's only needed with structs.
  - You can change any variable property on a class even when the class is created as a constant



**NOTE: if you don't want the ability to change variable property, make the property constant instead

Struct	Class
If changing one part of a struct effectively means destroying and recreating the entire struct	Classes don’t work this way: you can change any part of their properties without having to destroy and recreate the value. As a result, constant classes can have their variable properties changed freely.

That's why structs don’t allow their variable properties to be changed – it would mean destroying and recreating something that is supposed to be constant, which isn’t possible.	