Reviews

Properties

=> create method inside struct

Exemple :

```
struct Person {
    var clothes: String
    var shoes: String

    func describe() {
        print("I like wearing \(clothes) with \(shoes)")
    }
}

let taylor = Person(clothes: "T-shirts", shoes: "sneakers")
let other = Person(clothes: "short skirts", shoes: "high heels")
taylor.describe()
other.describe()
```

NB : As you can see, when you use a property inside a method it will automatically use the value that belongs to the same object.

Properties observers (willSet / didSet)

=> Swift lets you add code to be run when a property is about to be changed or has been changed.

=> This is frequently a good way to have a user interface update when a value changes.

Two kinds of property observer: **willSet** and **didSet**, and they are called before or after a property is changed. In **willSet** Swift provides your code

with a special value called **newValue** that contains what the new property value is going to be, and in **didSet** you are given **oldValue** to represent the previous value.
Let's attach two property observers to the **clothes** property of a **Person** struct:

Exemple :

```
struct Person {
    var clothes: String {
        willSet {
            updateUI(msg: "I'm changing from \(clothes) to \(newValue)")
        }

        didSet {
            updateUI(msg: "I just changed from \(oldValue) to \(clothes)")
        }
    }
}

func updateUI(msg: String) {
    print(msg)
}

var taylor = Person(clothes: "T-shirts")
taylor.clothes = "short skirts"
```

Computed properties

=> It's possible to make properties that are actually code behind the scenes. We already used the **uppercased()** method of strings, for example, but there’s also a property called **capitalized** that gets calculated as needed, rather than every string always storing a capitalized version of itself.

=> To make a computed property, place an open brace after your property then use either **get** or **set** to make an action happen at the appropriate time. For example, if we wanted to add a **ageInDogYears** property that automatically returned a person's age multiplied by seven, we'd do this:

Exemple :

```
struct Person {
    var age: Int

    var ageInDogYears: Int {
        get {
            return age * 7
        }
    }
}

var fan = Person(age: 25)
print(fan.ageInDogYears)
```

NB: If you intend to use them only for *reading* data you can just remove the get part entirely, like this:

```
var ageInDogYears: Int {
    return age * 7
}
```

Static properties and methods (shared properties or methods)

=> you access the property by using the full name of the type with static

Exemple :

```
struct TaylorFan {
    static var favoriteSong = "Look What You Made Me Do"

    var name: String
    var age: Int
}

let fan = TaylorFan(name: "James", age: 25)
```

```
print(TaylorFan.favoriteSong)
```

Access control

=> Access control lets you specify what data inside structs and classes should be exposed to the outside world, and you get to choose four modifiers:

- Public: this means everyone can read and write the property.
- Internal: this means only your Swift code can read and write the property. If you ship your code as a framework for others to use, they won’t be able to read the property.
- File Private: this means that only Swift code in the same file as the type can read and write the property.
- Private: this is the most restrictive option, and means the property is available only inside methods that belong to the type, or its extensions.

Exemple :

```
class TaylorFan {
   private var name: String?
}
```

Polymorphism and typecasting

=> classes can inherit from each other

```
      class Album {
   var name: String

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

```
class StudioAlbum: Album {
    var studio: String

    init(name: String, studio: String) {
        self.studio = studio
        super.init(name: name)
    }
}

class LiveAlbum: Album {
    var location: String

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

=> That defines three classes: albums, studio albums and live albums, with the latter two both inheriting from **Album**

=> var taylorSwift = StudioAlbum(name: "Taylor Swift", studio: "The Castles Studios")
var fearless = StudioAlbum(name: "Speak Now", studio: "Aimeeland Studio")
var iTunesLive = LiveAlbum(name: "iTunes Live from SoHo", location: "New York")

var allAlbums: [Album] = [taylorSwift, fearless, iTunesLive]

=> There we create an array that holds only albums, but put inside it two studio albums and a live album. This is perfectly fine in Swift because they are all descended from the **Album** class, so they share the same basic behavior.

```
class Album {
    var name: String

    init(name: String) {
        self.name = name
    }
    func getPerformance() -> String {
        return "The album \(name) sold lots"
    }
}

class StudioAlbum: Album {
    var studio: String

    init(name: String, studio: String) {
        self.studio = studio
        super.init(name: name)
    }

    override func getPerformance() -> String {
        return "The studio album \(name) sold lots"
    }
}

class LiveAlbum: Album {
    var location: String

    init(name: String, location: String) {
        self.location = location
        super.init(name: name)
    }

    override func getPerformance() -> String {
        return "The live album \(name) sold lots"
    }
}
```

=> The **getPerformance()** method exists in the **Album** class, but both child classes override it. When we create an array that holds **Albums**, we're actually filling it with subclasses of albums: **LiveAlbum** and **StudioAlbum**. They go into the array just fine because they inherit from the **Album** class, but they never lose their original class. So, we could write code like this:

```
var taylorSwift = StudioAlbum(name: "Taylor Swift", studio: "The Castles
Studios")
var fearless = StudioAlbum(name: "Speak Now", studio: "Aimeeland
Studio")
var iTunesLive = LiveAlbum(name: "iTunes Live from SoHo", location:
"New York")

var allAlbums: [Album] = [taylorSwift, fearless, iTunesLive]

for album in allAlbums {
   print(album.getPerformance())
}
```

NB : That will automatically use the override version of getPerformance() depending on the subclass in question. That's polymorphism in action: an object can work as its class and its parent classes, all at the same time.

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

Converting types with typecasting

Exemple :

```
for album in allAlbums {
   print(album.getPerformance())
}
```

That was our loop from a few minutes ago. The **allAlbums** array holds the type **Album**, but we know that really it's holding one of the subclasses: **StudioAlbum** or **LiveAlbum**. Swift doesn't know that, so if you try to write something like **print(album.studio)** it will refuse to build because only **StudioAlbum** objects have that property.

Typecasting in Swift comes in three forms, but most of the time you'll only meet two: **as?** and **as!**, known as optional downcasting and forced downcasting.

- The former means "I think this conversion might be true, but it might fail,"
- the second means "I know this conversion is true, and I'm happy for my app to crash if I'm wrong."

=> Swift treats the object – you're telling Swift that an object it thought was type A is actually type E.

```
for album in allAlbums {
    let studioAlbum = album as? StudioAlbum
}
```

THE SAME AS THAT =>

Exemple :

```
for album in allAlbums {
    print(album.getPerformance())

    if let studioAlbum = album as? StudioAlbum {
        print(studioAlbum.studio)
    } else if let liveAlbum = album as? LiveAlbum {
        print(liveAlbum.location)
    }
}
```

Swift will make **studioAlbum** have the data type **StudioAlbum?**. That is, an optional studio album: the conversion might have worked, in which case you have a studio album you can work with, or it might have failed, in which case you have nil.
This is most commonly used with **if let** to automatically unwrap the optional result, like this:

Closures

=> A closure can be thought of as a variable that holds code

NB : You never *need* to design your own closures

Exemple :

```
let vw = UIView()

UIView.animate(withDuration: 0.5, animations: {
    vw.alpha = 0
})
```

UIView is an iOS data type in UIKit that represents the most basic kind of user interface container. Don't worry about what it does for now, all that matters is that it's the basic user interface component. **UIView** has a method called **animate()** and it lets you change the way your interface looks using animation – you describe what's changing and over how many seconds, and Cocoa Touch does the rest.

Trailing closures

As closures are used so frequently, Swift can apply a little syntactic sugar to make your code easier to read. The rule is this: if the last parameter to a method takes a closure, you can eliminate that parameter and instead provide it as a block of code inside braces. For example, we could convert the previous code to this:

```
let vw = UIView()

UIView.animate(withDuration: 0.5) {
    vw.alpha = 0
}
```