X / Protocols and extensions

Protocols

=> Protocols let us define how structs, classes, and enums ought to work: what methods they should have, and what properties they should have

=> Protocols let you describe what methods something should have, but don't provide the code inside (* compare to Extensions)

Exemple :

```
struct Book {
    var name: String
}

func buy(_ book: Book) {
    print("I'm buying \(book.name)")
}
```

=> Exemple : protocol-based approach we would first create a protocol that declares the basic functionality we need

```
protocol Purchaseable {
    var name: String { get set }
}
```

 - - - - - - - - - - -

=> define as many structs as we need, with each one conforming to that protocol by having a name string :

```
struct Book: Purchaseable {
    var name: String
    var author: String
}
```

```swift
struct Movie: Purchaseable {
   var name: String
   var actors: [String]
}

struct Car: Purchaseable {
   var name: String
   var manufacturer: String
}

struct Coffee: Purchaseable {
   var name: String
   var strength: Int
}
```

 - - - - - - - -

=> we can rewrite the buy() function so that it accepts any kind of Purchaseable item:

```swift
func buy(_ item: Purchaseable) {
   print("I'm buying \(item.name)")
}
```

Exemple 2 :

```swift
protocol Purchaseable {
      var price: Double { get set }
      var currency: String { get set }
}

protocol Climbable {
      var height: Double { get }
      var gradient: Int { get }
}
```

Total score: 12/12 checked

Protocol inheritance

=> One protocol can inherit from another in a process known as *protocol inheritance*

Exemple :

three protocols:

- **Payable** requires conforming types to implement a **calculateWages()** method,
- **NeedsTraining** requires conforming types to implement a **study()** method,
- **HasVacation** requires conforming types to implement a **takeVacation()** method

```
protocol Payable {
   func calculateWages() -> Int
}

protocol NeedsTraining {
   func study()
}

protocol HasVacation {
   func takeVacation(days: Int)
}
```

 - - - - - - - -

=> We can create a single Employee protocol that brings them together in one protocol

```
protocol Employee: Payable, NeedsTraining, HasVacation { }
```

NB : Reason for using protocol inheritance is to combine functionality for common work :

Exemple :

- All products have a price and a weight
- All computers have a CPU, plus how much memory they have and how much storage
- All laptops have a screen size

Complex exemple :

=> define a **Computer** protocol:

```
protocol Computer {
   var price: Double { get set }
   var weight: Int { get set }
   var cpu: String { get set }
   var memory: Int { get set }
   var storage: Int { get set }
}
```

=> Then define a **Laptop** protocol :

```
protocol Laptop {
   var price: Double { get set }
   var weight: Int { get set }
   var cpu: String { get set }
   var memory: Int { get set }
   var storage: Int { get set }
   var screenSize: Int { get set }
}
```

- - - - - - - - -

Simpler Exemple :

```
protocol Product {
   var price: Double { get set }
   var weight: Int { get set }
}


protocol Computer: Product {
   var cpu: String { get set }
   var memory: Int { get set }
   var storage: Int { get set }
}
```

=> define what a **Laptop** looks like, by basing it on a **Computer** (and therefore also a **Product**

```
protocol Laptop: Computer {
   var screenSize: Int { get set }
}
```

NB : using protocol inheritance in this way lets share definitions and reduce duplication

Total score: 12/12 checked

Extensions

=> Extensions let us add functionality to classes, structs, and more, which is helpful for modifying types we don’t own

=> Extensions let you provide the code inside your methods, but only affect one data type

Exemple 1 :

```
extension Int {
   func squared() -> Int {
      return self * self
   }
}

let number = 8
number.squared() <- return 64
```

Exemple 2 :

```
extension String {
      func isUppercased() -> Bool {
            return self == self.uppercased()
      }
}
```

Total score: 12/12 checked

Protocol extensions

=> they are like regular extensions, except rather than extending a specific type like Int you extend a whole protocol so that all conforming types get your changes

Exemple :

```
let pythons = ["Eric", "Graham", "John", "Michael", "Terry", "Terry"]
let beatles = Set(["John", "Paul", "George", "Ringo"])
```

 - - - - - -
=> we can write an extension to that protocol to add a **summarize()** method to print the collection neatly

```
extension Collection {
    func summarize() {
        print("There are \(count) of us:")

        for name in self {
            print(name)
        }
    }
}

pythons.summarize()
beatles.summarize()
```

NB : We use them to add functionality directly to protocols, which means we don't need to copy that functionality across many structs and classes

Exemple 1 :

=> Swift's arrays/dictionnaries/Sets, have an **allSatisfy()** method that returns true if all the items in the array pass a test.

```
let numbers = [4, 8, 15, 16]
let allEven = numbers.allSatisfy { $0.isMultiple(of: 2) }
let numbers2 = Set([4, 8, 15, 16])
let allEven2 = numbers2.allSatisfy { $0.isMultiple(of: 2) }

let numbers3 = ["four": 4, "eight": 8, "fifteen": 15, "sixteen": 16]
let allEven3 = numbers3.allSatisfy { $0.value.isMultiple(of: 2) }
```

Exemple 2 :

```
protocol Politician {
    var isDirty: Bool { get set }
    func takeBribe()
}
extension Politician {
    func takeBribe() {
        if isDirty {
            print("Thank you very much!")
        } else {
            print("Someone call the police!")
        }
    }
}
```

Exemple 3 :

```
protocol Anime {
    var availableLanguages: [String] { get set }
    func watch(in language: String)
}
extension Anime {
    func watch(in language: String) {
        if availableLanguages.contains(language) {
            print("Now playing in \(language)")
        } else {
            print("Unrecognized language.")
        }
    }
}
```

Exemple 4 :

```
protocol Bartender {
	func makeDrink()
}
extension Bartender {
	func makeDrink(name: String) {
		print("One \(name) coming right up.")
	}
}
```

Exemple 5 :

```
protocol Hamster {
	var name: String { get set }
	func runInWheel(minutes: Int)
}
extension Hamster {
	func runInWheel(minutes: Int) {
		print("\(name) is going for a run.")
		for _ in 0..<minutes {
			print("Whirr whirr whirr")
		}
	}
}
```

Total score: 12/12 checked

Protocol-oriented programming

=> crafting your code around protocols and protocol extensions.
=> All properties Protocol must have { get } or { get set } after them.

Exemple :

=> First, call a protocol **Identifiable** that requires any conforming type to have an **id** property and an **identify()** method :

```
protocol Identifiable {
    var id: String { get set }
    func identify()
}
```

=> protocol extensions allow us to provide a default:

```
extension Identifiable {
    func identify() {
        print("My ID is \(id).")
    }
}
```

=> When we create a type that conforms to **Identifiable** it gets **identify()** automatically

```
struct User: Identifiable {
    var id: String
}

let twostraws = User(id: "twostraws")
twostraws.identify()                    <- print automatically -> My ID is twostraws.
```

Exemple 2 :

```
protocol SuitableForKids {
        var minimumAge: Int { get set }
        var maximumAge: Int { get set }
}
protocol SupportsMultiplePlayers {
        var minimumPlayers: Int { get set }
        var maximumPlayers: Int { get set }
}
struct FamilyBoardGame: SuitableForKids, SupportsMultiplePlayers {
        var minimumAge = 3
        var maximumAge = 110
```

```
    var minimumPlayers = 1
    var maximumPlayers = 4
}
```

NB :

Some might say that the only real difference between the two is that in protocol-oriented programming (POP) we prefer to build functionality by composing protocols (“this new struct conforms to protocols X, Y, and Z”), whereas in object-oriented programming (OOP) we prefer to build functionality through class inheritance.

However, even that is dubious because OOP developers also usually prefer composing functionality to inheriting it – it’s just easier to maintain.

In fact, the only important difference between the two is one of mindset: POP developers lean heavily on the protocol extension functionality of Swift to build types that get a lot of their behavior from protocols.

This makes it easier to share functionality across many types, which in turn lets us build bigger, more powerful software without having to write so much code.

Total score: 6/6 checked

Protocols and extensions summary

1. Protocols describe what methods and properties a conforming type must have, but don’t provide the implementations of those methods.

2. You can build protocols on top of other protocols, similar to classes.

3. Extensions let you add methods and computed properties to specific types such as **Int**.

4. Protocol extensions let you add methods and computed properties to protocols.

5. Protocol-oriented programming is the practice of designing your app architecture as a series of protocols, then using protocol extensions to provide default method implementations.