Class 10

CSC 496: iOS App Development
Swift Fundamentals: Classes (2), Protocols

Si Chen (schen@wcupa.edu)

Review

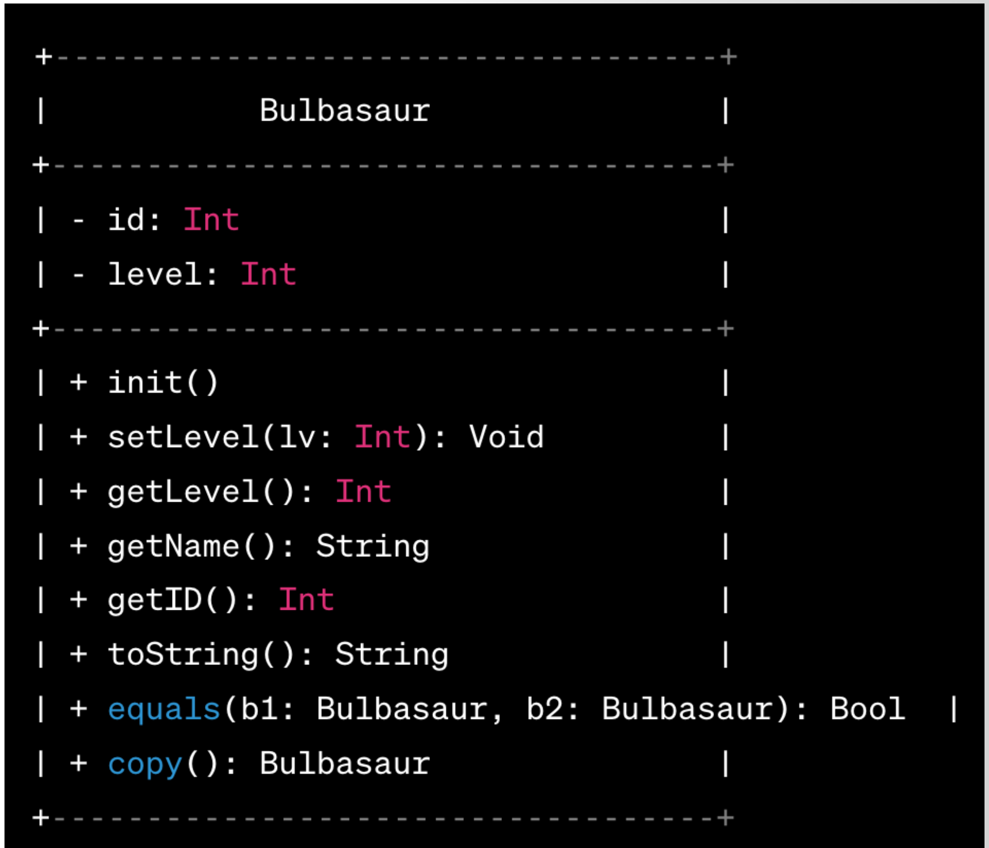
Bulbasaur Class

Bulbasaur, known as Fushigidane in Japan, is a Pokemon species in Nintendo and Game Freak's Pokemon franchise. Designed by Ken Sugimori, their name is a combination of the words ``bulb" and ``dinosaur". First appearing in Pokemon Red and Blue.

Exercise: Bulbasaur Class

We know that **Bulbasaur** evolves into **Ivysaur** when it reaches **level 16** and subsequently transforms into **Venusaur** upon attaining **level 32**.

UML Diagram for Bulbasaur Class
Please refer to the attached UML diagram to design a class for Bulbasaur (Bulbasaur.swift):



```
+------------------------------------+
|            Bulbasaur               |
+------------------------------------+
| - id: Int                          |
| - level: Int                       |
+------------------------------------+
| + init()                           |
| + setLevel(lv: Int): Void          |
| + getLevel(): Int                  |
| + getName(): String                |
| + getID(): Int                     |
| + toString(): String               |
| + equals(b1: Bulbasaur, b2: Bulbasaur): Bool  |
| + copy(): Bulbasaur                |
+------------------------------------+
```

Bulbasaur Class

Attributes for Each Bulbasaur Object

•**id**: Represents the evolutionary stage of the Pokemon. It is set to 1 for Bulbasaur, 2 for Ivysaur, and 3 for Venusaur. The initial value is 1.
•**level**: Denotes the current level of the Bulbasaur object, initialized to 1.

Methods for Each Bulbasaur Object

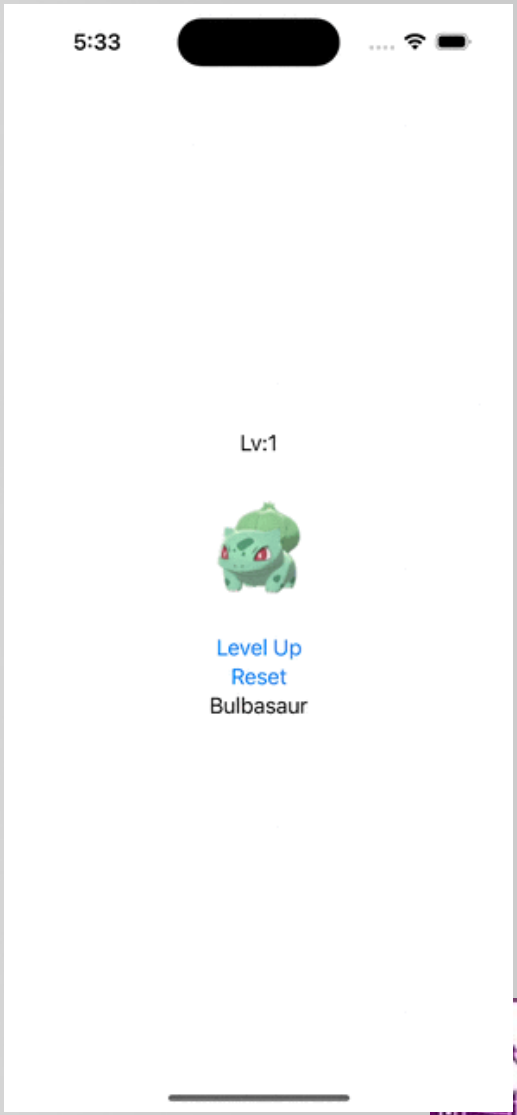
•**setLevel(lv: int): void**: Takes an integer value as an argument to update the object's level. If the new level falls within the range [16-31], the Pokemon evolves into Ivysaur. If the level is 32 or higher, it evolves into Venusaur.
•**getLevel(): int**: Returns an integer representing the current level of the Pokemon.
•**getName(): String**: Returns the name of the Pokemon as a string ("Bulbasaur" for id = 1, "Ivysaur" for id = 2, and "Venusaur" for id = 3).
•**getID()**: Returns the current id value.
•**toString()**: Outputs the current level and id of the Bulbasaur object.
•**equals()**: Compares the level and id of two different Bulbasaur objects to determine if they are equal.
•**copy()**: Creates a clone of a Bulbasaur object with the same level and id values.

Exercise: Bulbasaur Level-Up App

Objective:
Use the **Bulbasaur class** and design a SwiftUI iOS application that manages the state of a Bulbasaur as it levels up and evolves.

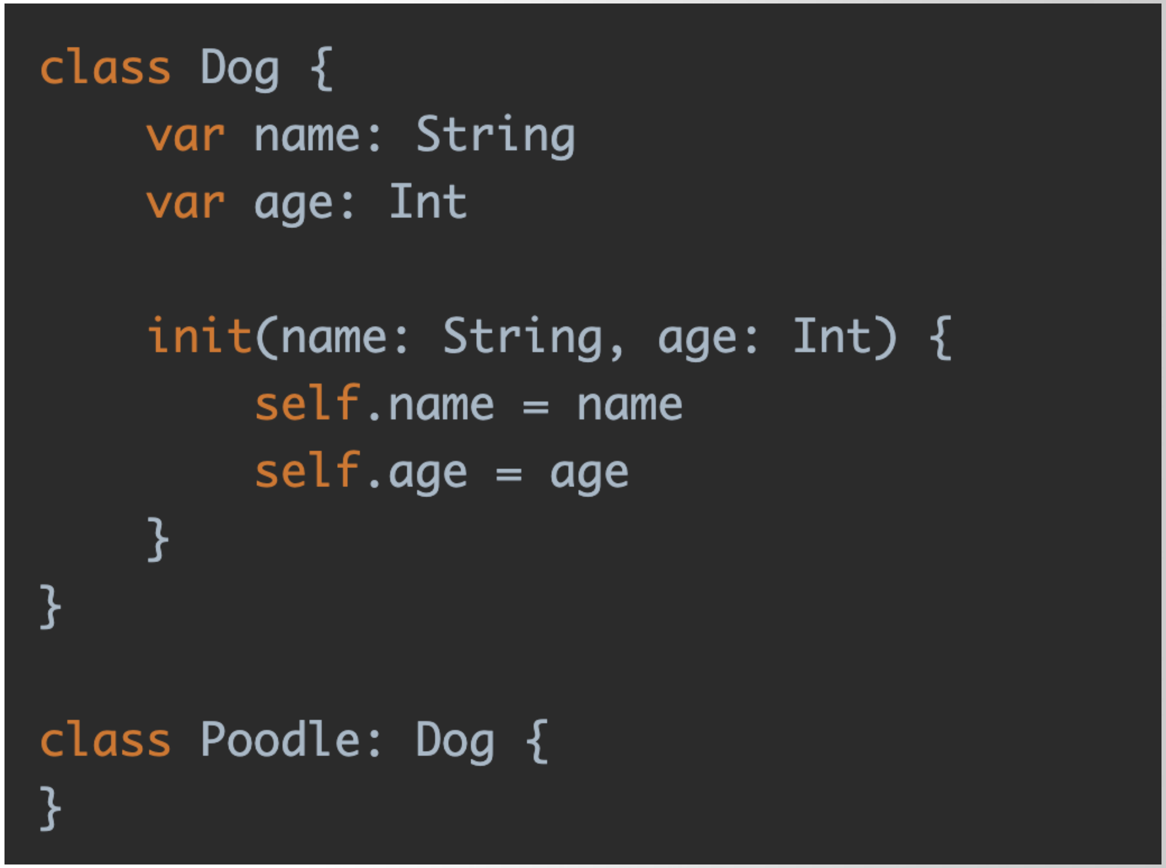
Requirements:
1.UI Elements:
The application should have the following UI elements:

1. A text label displaying the current level of the Pokémon (e.g., "Lv: 1").
2. An image displaying the current form of the Pokémon (Bulbasaur, Ivysaur, or Venusaur). The image assets for these forms are already provided in the course website, please download and add them to your project. The filename for each image corresponds to the Pokémon's name (e.g., "Bulbasaur.png", "Ivysaur.png", "Venusaur.png").
3. A button labeled "Level Up" that, when pressed, increments the Pokémon's level by 1 and updates its form if it evolves.
4. A ”reset” button to reset level back to 1



Inheritance and Polymorphism

- The Base class, also known as the superclass or parent class, is a class that is being inherited from.
- The Derived class, also known as subclass or child class, is a class that inherits from another class. Here's an example of a base class "Dog" and a derived class "Poodle".



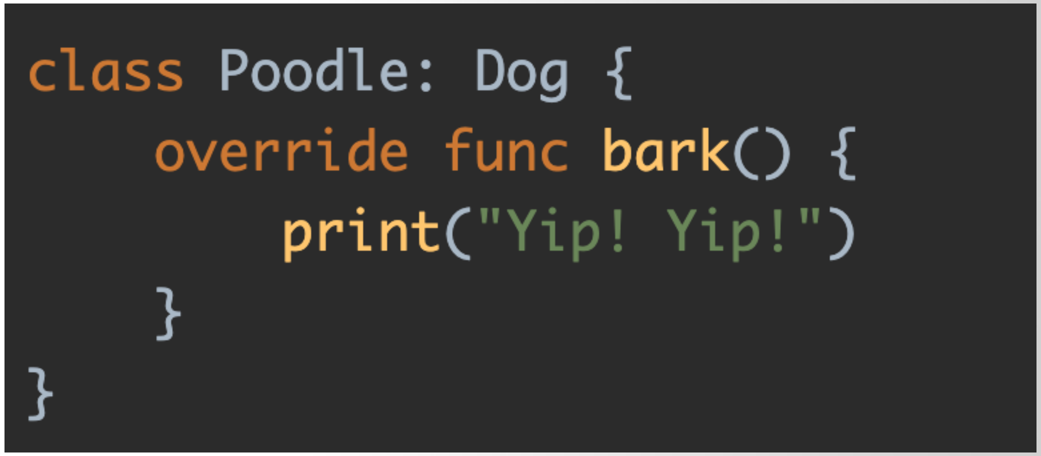
```
class Dog {
    var name: String
    var age: Int

    init(name: String, age: Int) {
        self.name = name
        self.age = age
    }
}

class Poodle: Dog {
}
```

Method Overriding

- Here, we have overridden the **bark** method for the **Poodle** class. Now, when we call the **bark** method on a **Poodle** object, it will print "Yip! Yip!" instead of a regular "Woof!".



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

Polymorphism

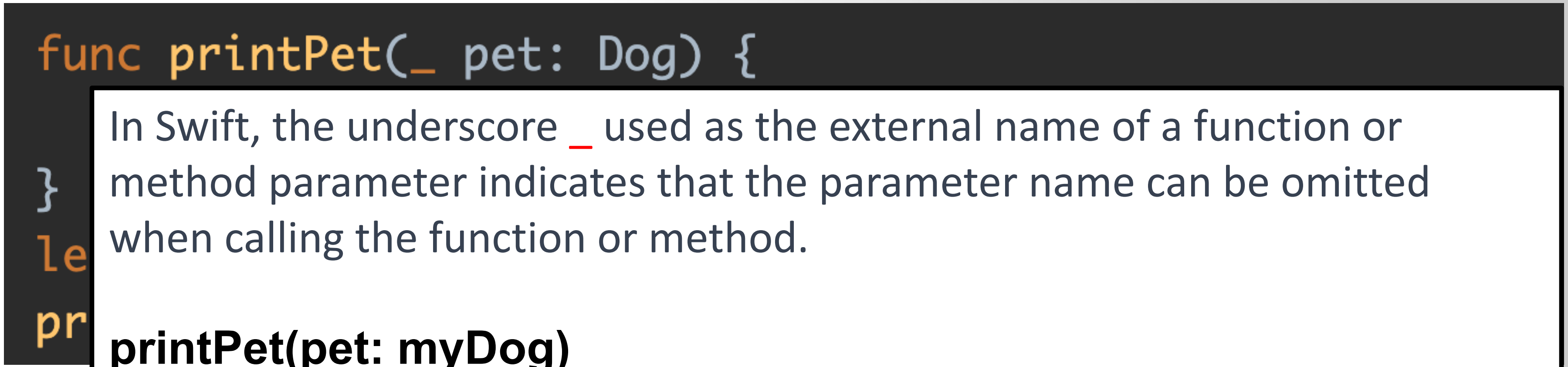
Polymorphism means the ability to request that the same operations be performed by a wide range of different types of things.

```swift
func printPet(_ pet: Dog) {
    print("\(pet.name) is a \(pet.age) year old pet.")
}
let aPoodle = Poodle(name: "Fifi", age: 5)
printPet(aPoodle)
```

Even though we're passing a **Poodle**, our **printPet** function treats it like a **Dog**. This is polymorphism in action!

Polymorphism

Polymorphism means the ability to request that the same operations be performed by a wide range of different types of things.



```
func printPet(_ pet: Dog) {

}
le
pr
```

In Swift, the underscore _ used as the external name of a function or method parameter indicates that the parameter name can be omitted when calling the function or method.

printPet(pet: myDog)

Concise way (if use _ as the external name): **printPet(myDog)**

Type Casting and **Checking**

1. **Type Casting** is a way to check the type of an instance, or to treat that instance as a different superclass or subclass from somewhere else in its own class hierarchy. This is incredibly useful when working with classes and subclasses in Swift. Firstly, we need to understand two fundamental Swift operators:
   - **"is" operator**: Tests the type of a class instance. Think of it like you're asking "***is*** this a certain type?"
   - **"as" operator**: Performs certain cast to a class instance.

Downcasting

- Downcasting can be either:
  - Safe (optional) - Using **"as?"**: This is when you're not sure if the downcast will succeed. The result is an optional value that will be nil if the downcast was not possible.
  - Forced (non-optional) - Using **"as!"**: This is when you're sure that the downcast will always succeed. But be careful -- using this when the downcast won't succeed will cause a runtime error!

```swift
class Vehicle {}

class Car: Vehicle {
    func beep() {
        print("Beep Beep!")
    }
}

let myVehicle: Vehicle = Car()

// Safe downcasting
if let car = myVehicle as? Car {
    car.beep()
}

// Unsafe downcasting
let forcedDowncastCar = myVehicle as! Car
forcedDowncastCar.beep()
```

In this example, '**myVehicle**' is of type '**Vehicle**' but it refers to an instance of '**Car**'. The '**beep**' method is specific to the '**Car**' class. '**myVehicle**' cannot call '**beep**' unless it is downcast to '**Car**'.

Type Checking in Hierarchies

- With the "is" operator, we can check the type of an object. It returns true if the object is of that type. If the object's class is a child of that type, this will also return true. Here's a quick example:

```
let isVehicle = myVehicle is Vehicle // true
let isCar = myVehicle is Car // true
```

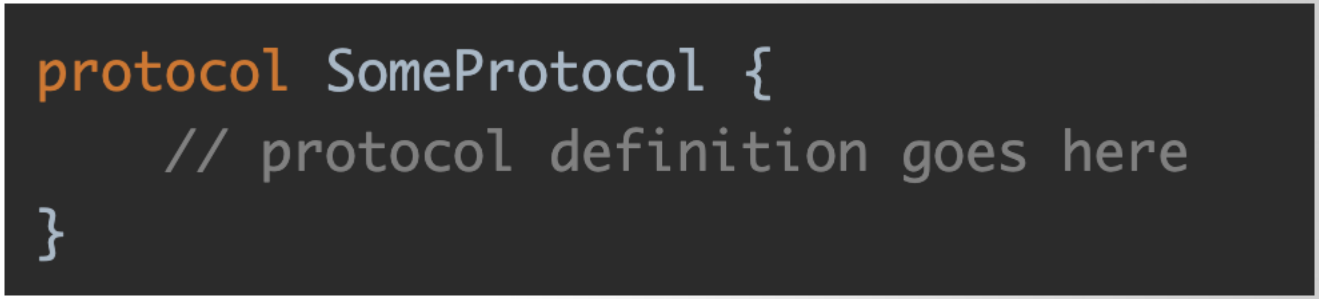
In the above example, **myVehicle** is both a **Vehicle** and a **Car**, so when we test it with the is operator, both checks return true.

Protocols

- Unlike a class, struct, or enum, a **protocol** is a blueprint of methods, properties, and other requirements that suit a particular action or piece of functionality.
- They're like the rules of the game, setting the standards for how classes, structures, and enumerations should appear and behave.

Protocol Syntax

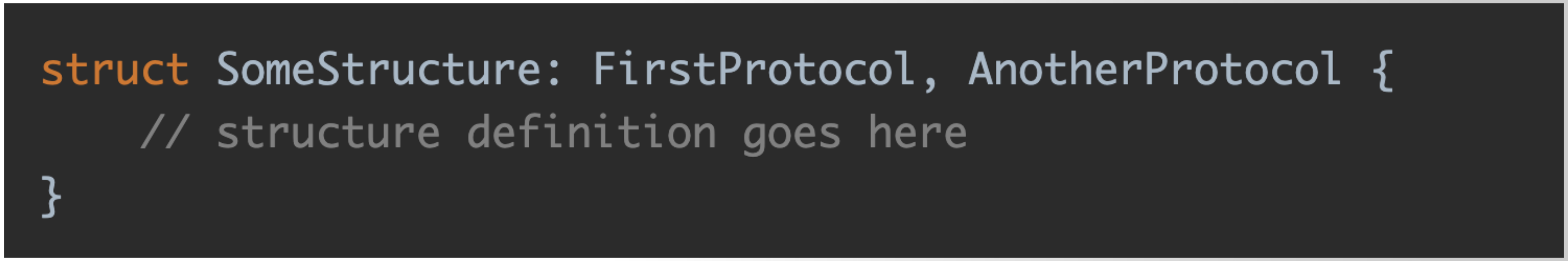
Swift protocols have a specific syntax, similar to what you would see when defining a class or structure. However, protocols won't provide any implementation for the requirements they define.



```
protocol SomeProtocol {
    // protocol definition goes here
}
```

Adopting Protocols

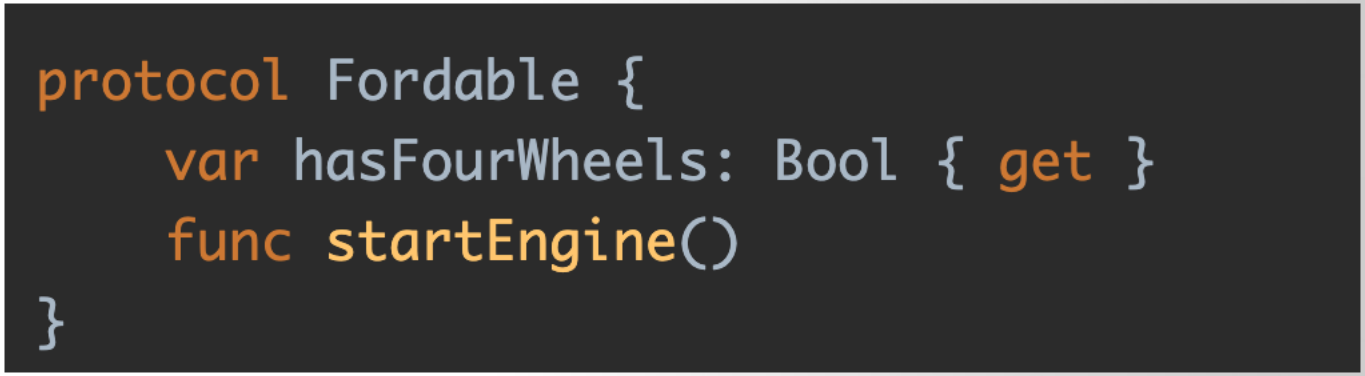
- A class, structure, or enumeration can adopt a protocol by listing its name after their own, separated by a colon :
- Look at this simple example:



```
struct SomeStructure: FirstProtocol, AnotherProtocol {
    // structure definition goes here
}
```

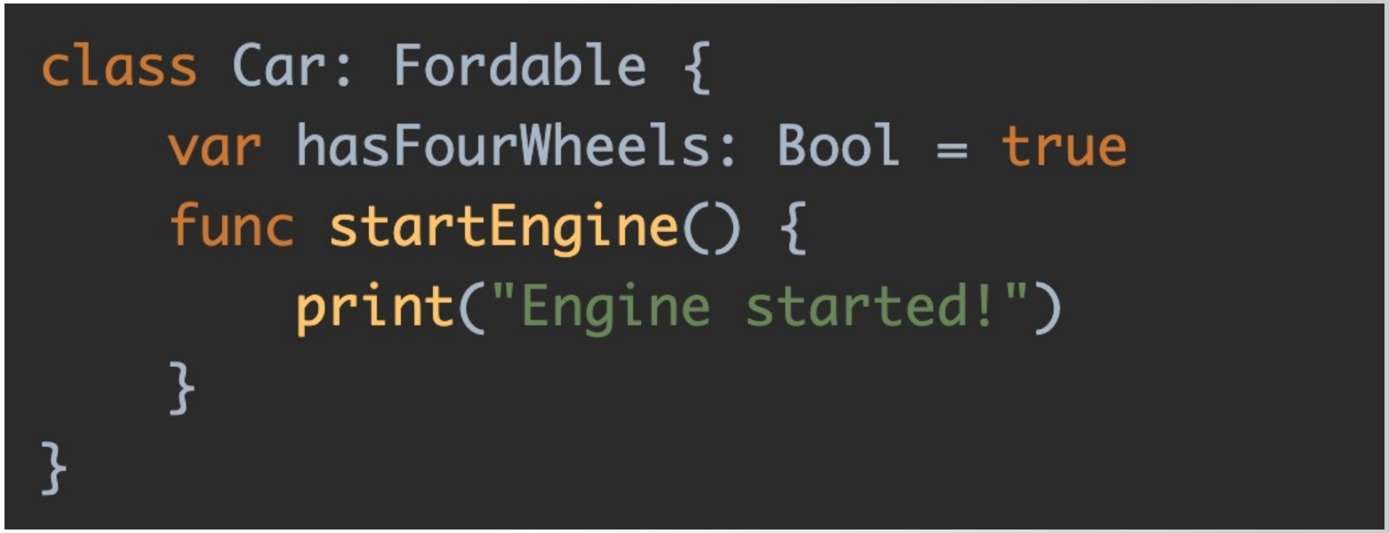
Here, 'SomeStructure' is adopting and conforming to 'FirstProtocol' and 'AnotherProtocol'.

Adopting Protocols: Example



```
protocol Fordable {
    var hasFourWheels: Bool { get }
    func startEngine()
}
```

In this protocol, **Fordable**, there are a variable **hasFourWheels** and a function **startEngine**(). Any type that adopts this protocol will need to implement these two properties.



```
class Car: Fordable {
    var hasFourWheels: Bool = true
    func startEngine() {
        print("Engine started!")
    }
}
```

Protocol Inheritance

- Swift protocols can inherit from one or many protocols. This enables you to build on top of existing protocols to provide more specialized behavior. Let's see how it's done.

```
protocol InheritingProtocol: SomeProtocol, AnotherProtocol {
    // protocol definition goes here
}
```

```
protocol Vehicle {
    var hasWheels: Bool { get }
}
protocol Fordable: Vehicle {
    var hasFourWheels: Bool { get }
}
```

In this scenario, Fordable is inheriting Vehicle protocol. Therefore, anyone who will adopt Fordable protocol will need to fulfill the requirements of Vehicle as well.

Optional Protocol Requirements

- In Swift, protocols can also have optional requirements, which are not required to be implemented by a type in order to conform to the protocol. These optional requirements are marked by the **optional** keyword.

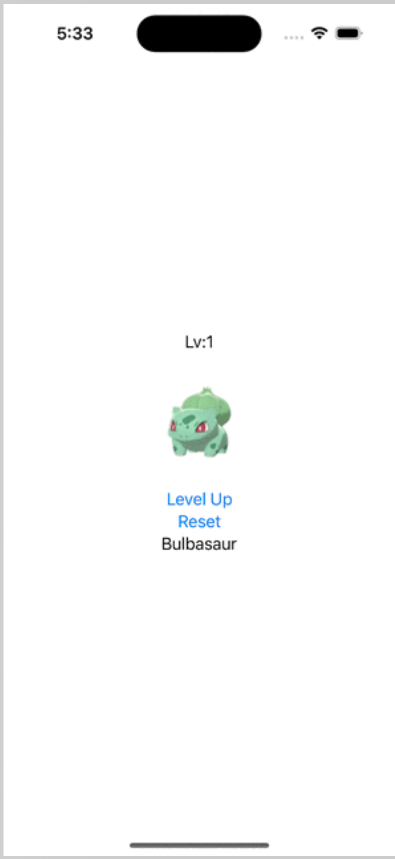
```
@objc protocol Moveable {
    @objc optional func startMoving()
    @objc optional func stopMoving()
}
```

- Here **startMoving**() and **stopMoving**() are not mandatory for any type that adopts **Moveable** protocol.

Understanding SwiftUI's Data Flow - ObservableObject and @Published

Scenario: You have a custom class, and you want your SwiftUI view to update when an attribute of the class changes.

Problem with Bulbasaur Class in SwiftUI: Simply updating the class attribute won't update the UI.



- **ObservableObject**: A protocol that SwiftUI uses to re-render a view when the data it observes changes.
- **@Published**: A property wrapper that marks properties of an ObservableObject to notify the UI of changes.

Understanding SwiftUI's Data Flow - ObservableObject and @Published

- **How to Implement ObservableObject**
  1. Conform your class to the **ObservableObject** protocol.
  2. Use **@Published** to annotate attributes that, when changed, should trigger a UI update.

```swift
class Bulbasaur: ObservableObject {
    // Attributes
    @Published var id: Int
    @Published var level: Int
```

Use @ObservedObject or @StateObject to bind an instance of your class to the view.

```swift
@StateObject var b = Bulbasaur()
```

•**@ObservedObject**: Use when your observable object is passed from a parent view.
•**@StateObject**: Use when your observable object is created within the view.

Q & A