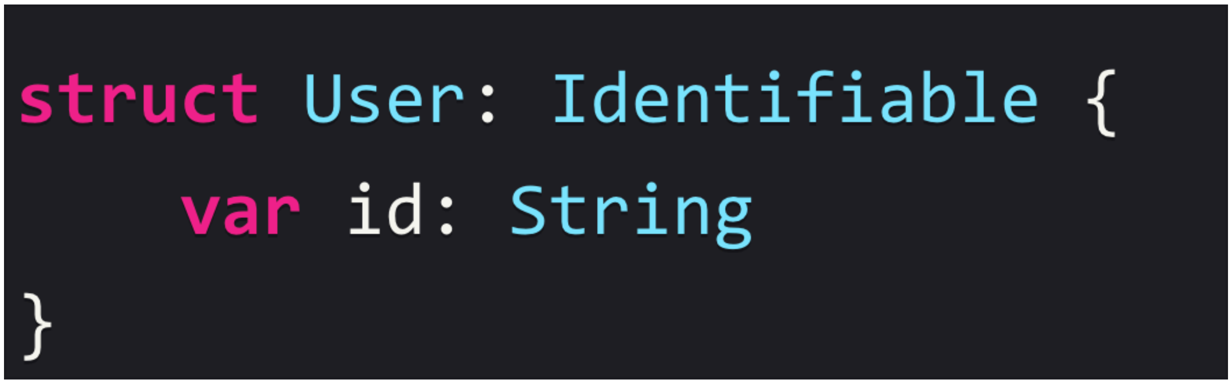
SWIFT INTRO : PROTOCOLS and EXTENSIONS

✓ Protocols

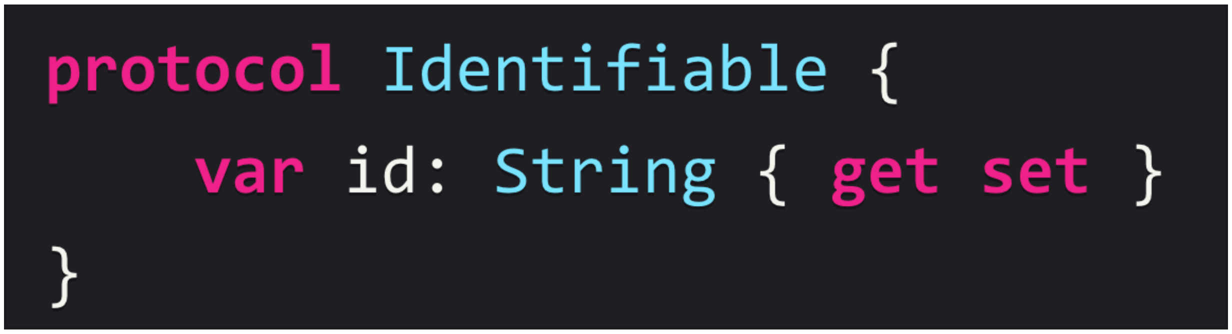
- Protocols are a way of describing what properties and methods something must have
- Protocols let us define how structs, classes, and enums ought to work: what methods they should have, and what properties they should have.
- We can't *create* instances of that protocol - it's a description of what we want, rather than something we can create and use directly.
    - But we *can* create a struct that conforms to it



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

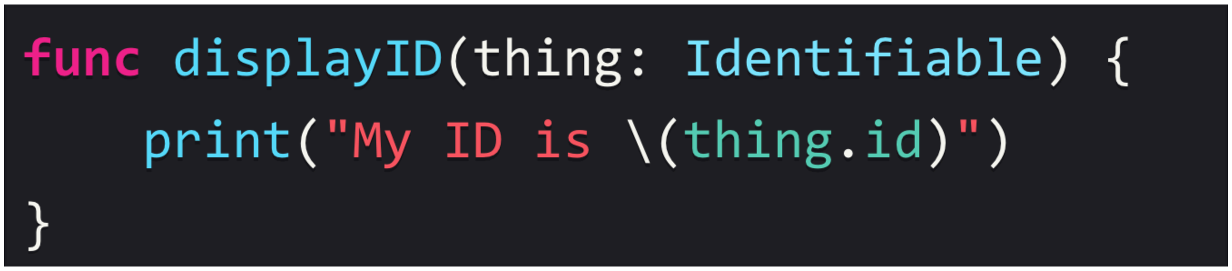
- **Swift's common protocols (we can create our own protocol as well):**
    - **Identifiable** protocol:

require all conforming types to have an **id** string that can be read ("get") or written ("set"):



```swift
protocol Identifiable {
    var id: String { get set }
}
```

we'll write a **displayID()** function that accepts any **Identifiable** object:



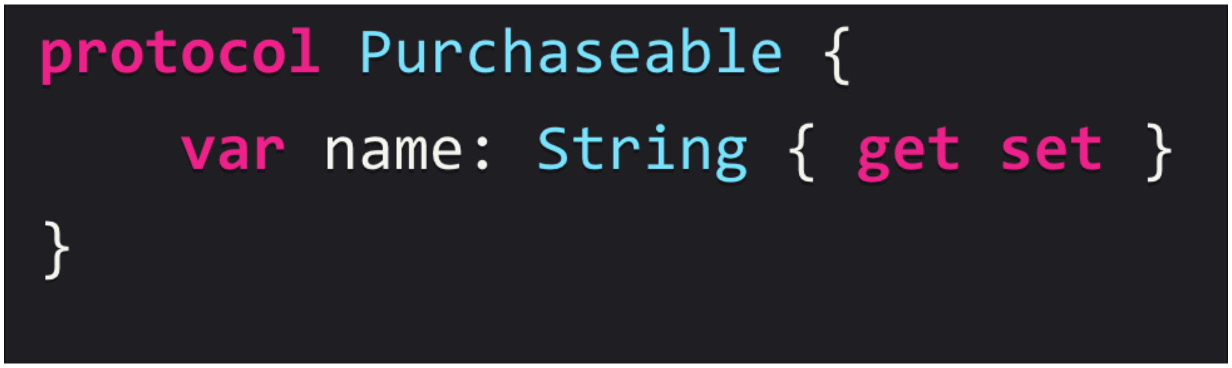
```swift
func displayID(thing: Identifiable) {
    print("My ID is \(thing.id)")
}
```

  - **Equatable**, which means “this thing can be checked for equality against other instances of the same type
  - **Comparable**, which means “this type can be checked to see whether it is less than or greater than another instance of the same type

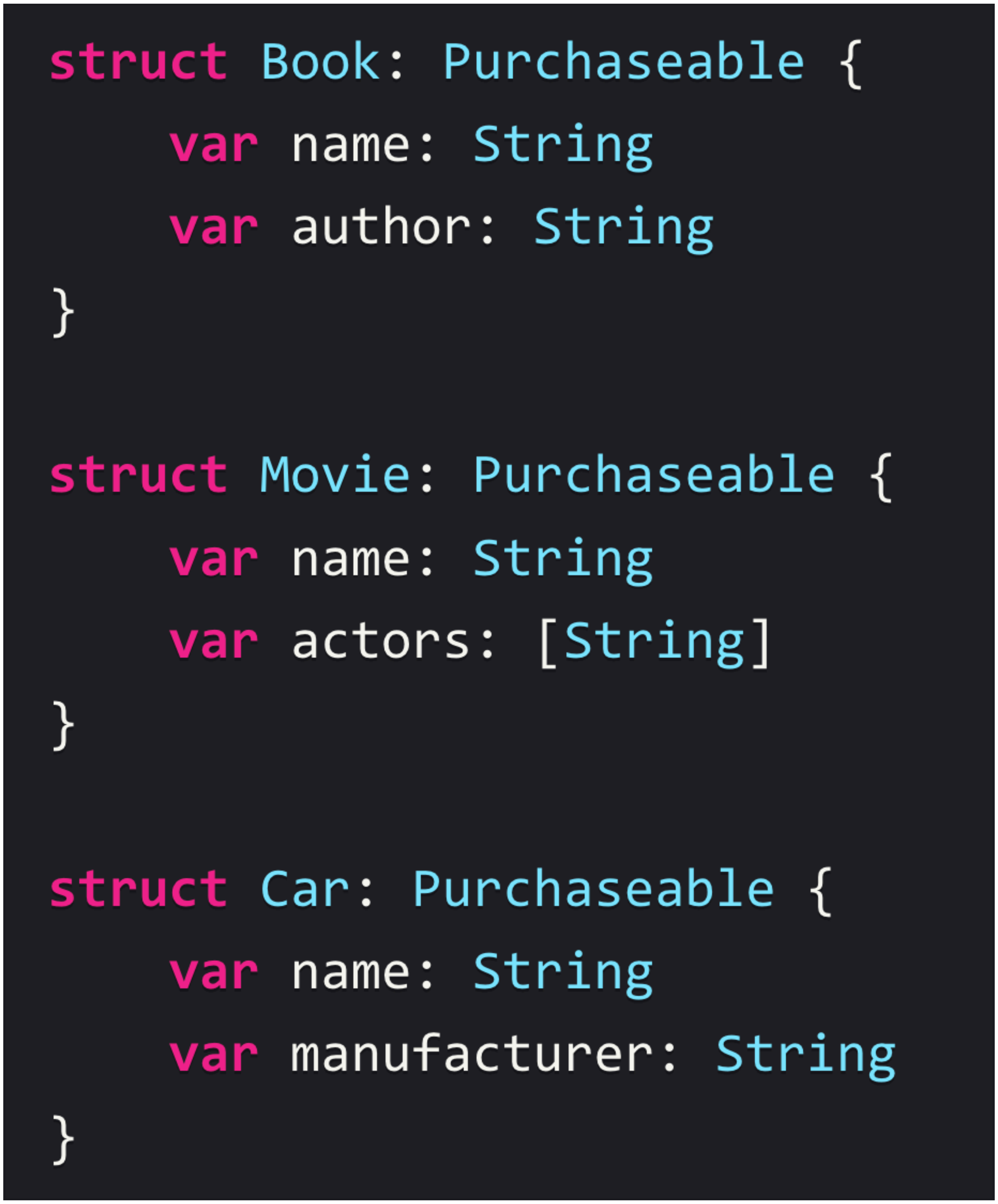
– Explain in general term:

this **buy()** method must accept a **Book** object,” we can say “this method can accept anything that conforms to the **Purchaseable** protocol. That might be a book, but it might also be a movie, a car, some coffee, and so on <= protocol

Ex:



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

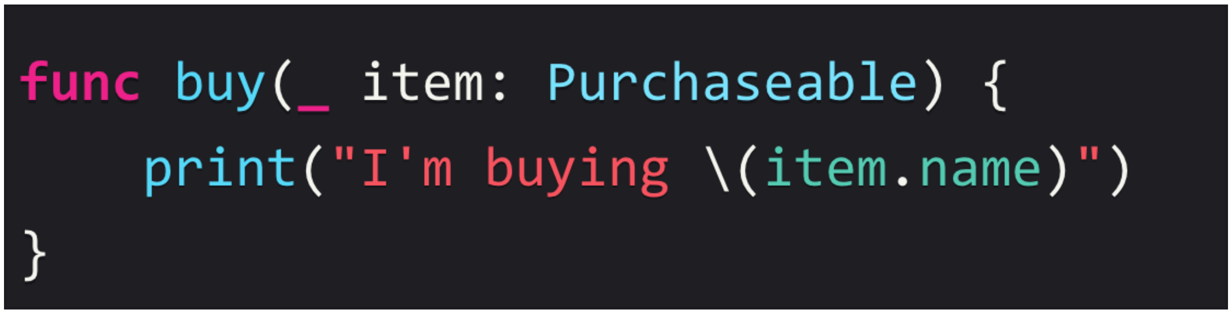


```swift
struct Book: Purchaseable {
    var name: String
    var author: String
}

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

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

=> we can write the **buy()** function so that it accepts any kind of **Purchaseable** item of any type (book, movie, car)



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

NOTE:
+ This guarantee that each **Purchaseable** item has a **name** property.
+ It *doesn't* guarantee that any of the other properties we defined will exist, only the ones that are specifically declared in the protocol.

✅ Protocol inheritance

- One protocol can inherit from another in a process known as ***protocol inheritance***.
- Unlike with classes, you can inherit from multiple protocols at the same time before you add your own customizations on top. (You can only inherit 1 class)
- Reason to use: is to combine functionality for common work => reduce duplication. For example:
  - 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

EX:
We have 3 protocols:

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

protocol NeedsTraining {
    func study()
}

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

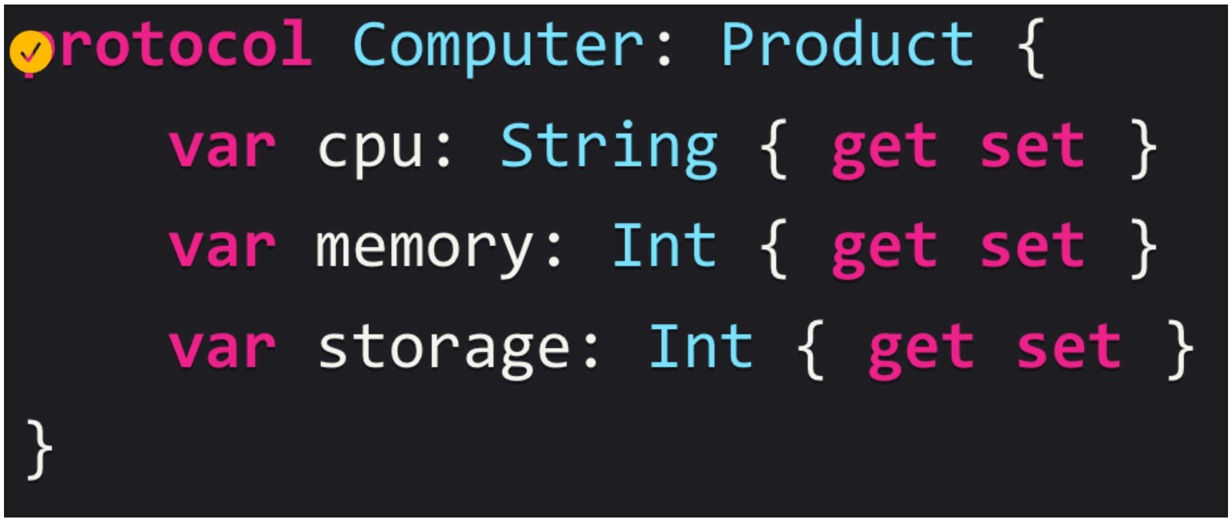
Now, a single **Employee** protocol that brings them together in one protocol.

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

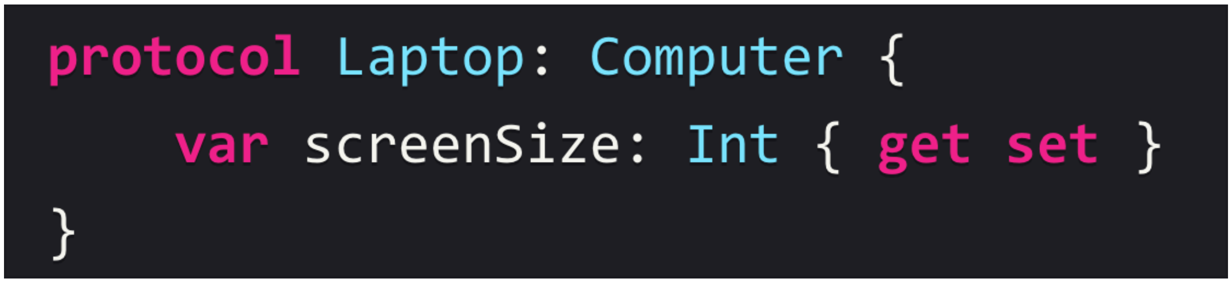
Another example:

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

Both Computer and Laptop are inherited from Product, so they will also have property price & weight which reduce duplication.



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

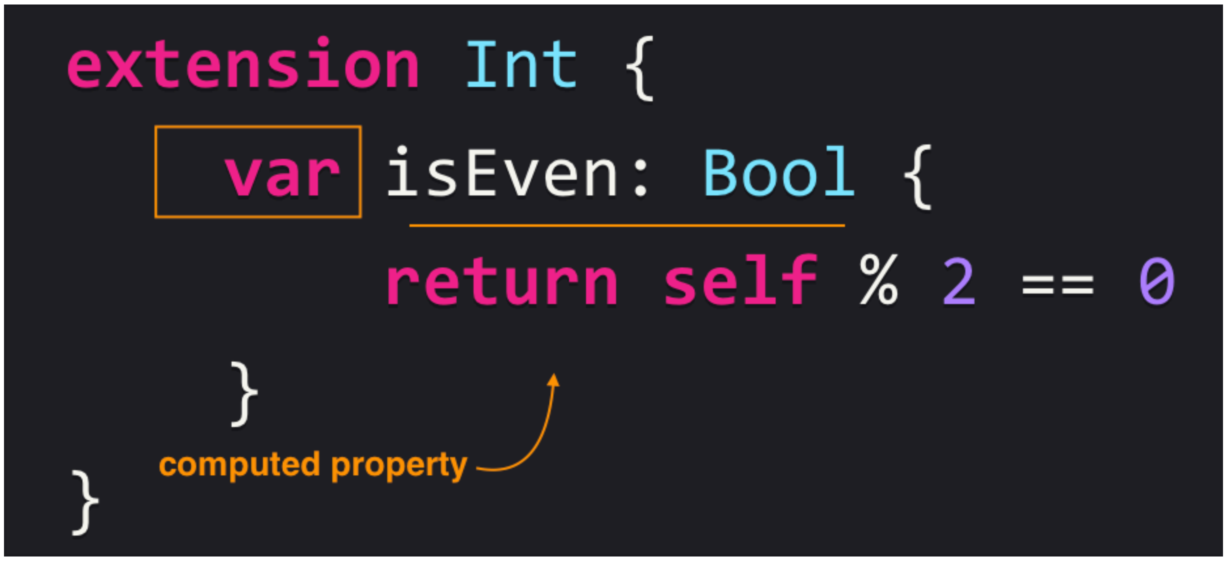


```swift
protocol Laptop: Computer {
    var screenSize: Int { get set }
}
```

Extension

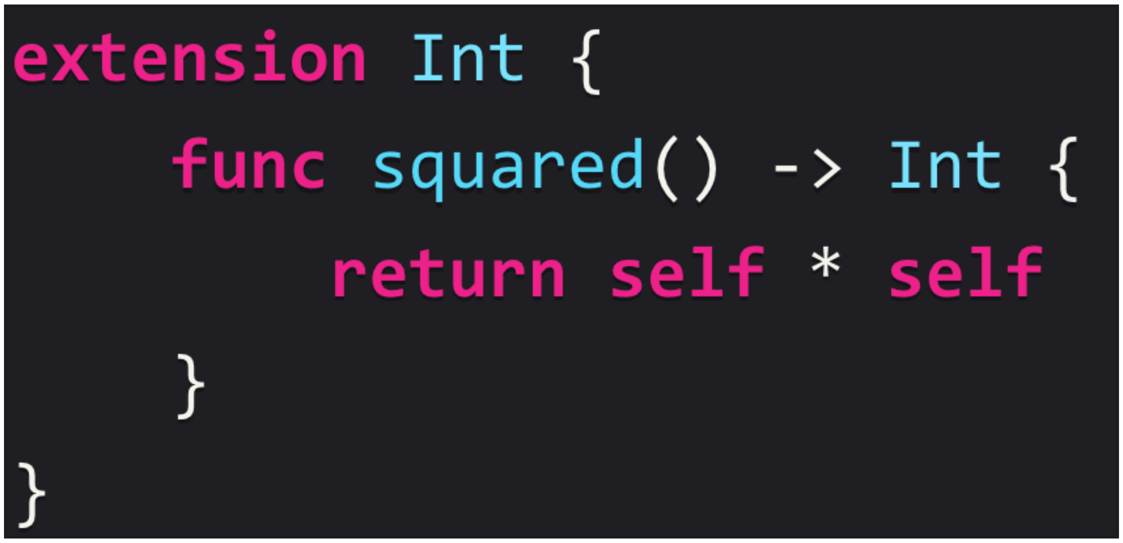
- Extension let us add functionality to classes, structs, and more, which is helpful for modifying types we don't own or make existing types do things they weren't originally designed to do.

- Methods added using extensions are indistinguishable from methods that were originally part of the type
- Swift doesn't let you add stored properties in extensions, so you **MUST use computed properties** instead.

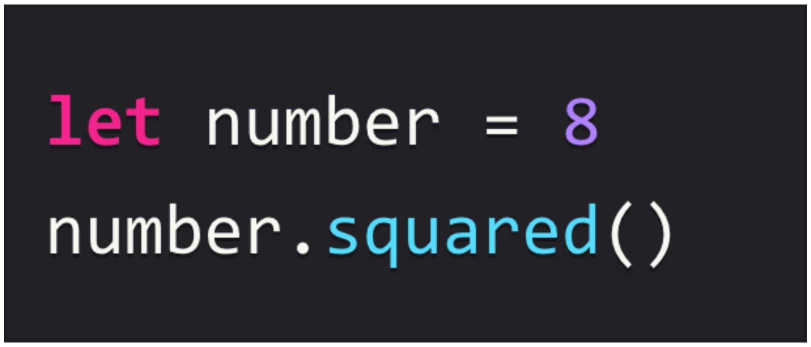


```swift
extension Int {
    var isEven: Bool {
        return self % 2 == 0
    }
}
```

EX: We could add an extension to the **Int** type so that it has
a **squared()** method that returns the current number multiplied by itself



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



```swift
let number = 8
number.squared()
```

- Extension is helpful for organizing code:
  - Conformance grouping

        - Means adding a protocol conformance to a type as an extension, adding all the required methods inside that extension.
    - Purpose grouping
        - Means creating extensions to do specific tasks, which makes it easier to work with large types.

- Protocol extension

Protocol	Extension	Protocol Extension
Protocols let you describe what methods something should have, but don't provide the code inside.	Extensions let you provide the code inside your methods, but only affect one data type – you can't add the method to lots of types at the same time.	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.

EX:
Here is an array and a set containing some names:

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

Arrays and sets both conform to a protocol called **Collection =>** write an extension to that protocol to add a **summarize()** method to print the collection

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

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

Now, Both **Array** and **Set** will now have that method

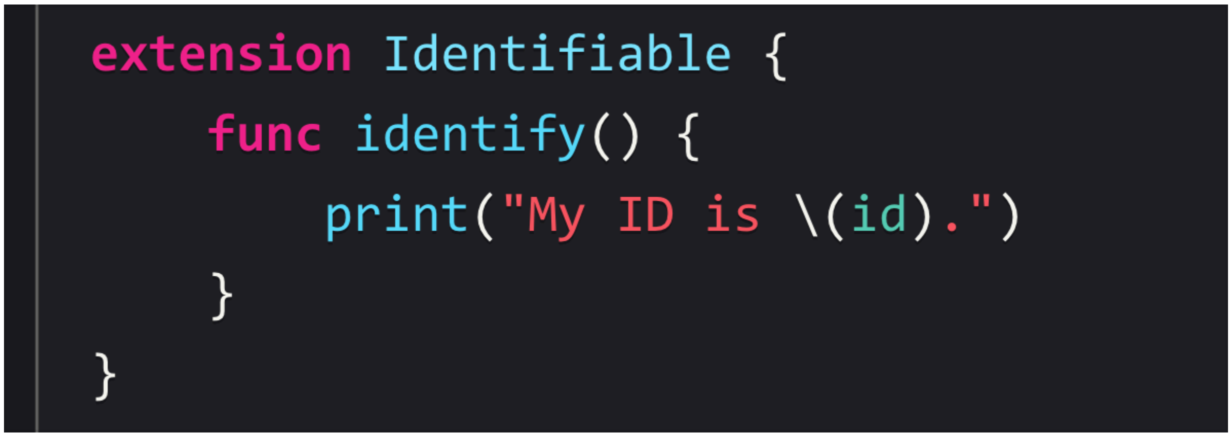
```swift
pythons.summarize()
beatles.summarize()
```

✅ Protocol-oriented programming:
A technique where crafting your code around protocols and protocol extensions

EX:

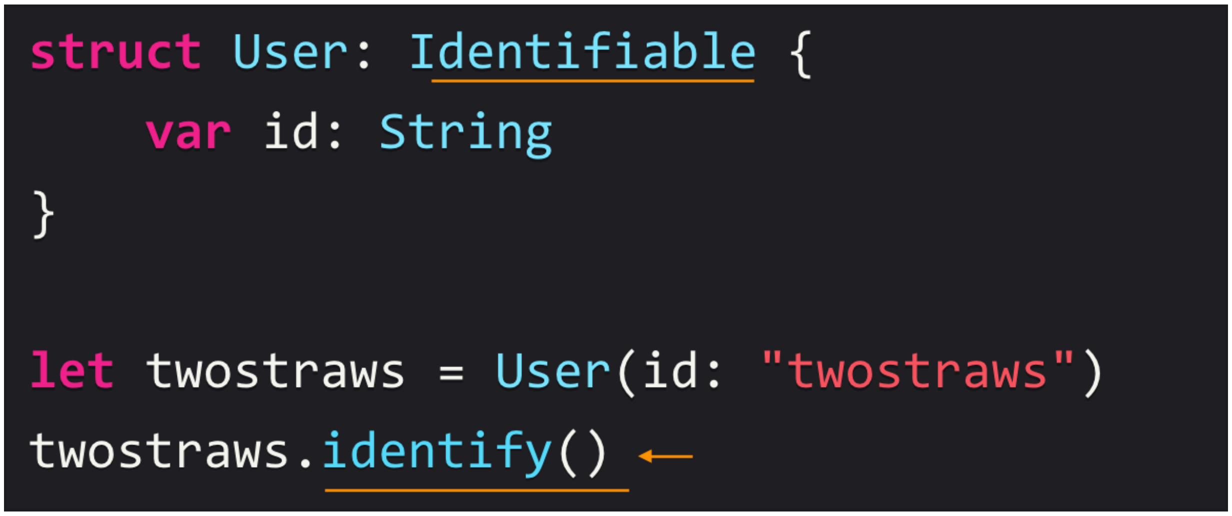
```swift
protocol Identifiable {
    var id: String { get set }
    func identify()
}
```

We *could* make every conforming type write their own **identify()** method, but protocol extensions allow us to provide a default:



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

Now when we create a type that conforms to **Identifiable** it gets **identify()** automatically:



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

let twostraws = User(id: "twostraws")
twostraws.identify()
```

Protocol-oriented programming	object-oriented programming
SAME:	both can place functionality into objects, use access control to limit where that functionality can be called, make one class inherit from another, and more.
DIFFERENT:	

we prefer to build functionality by composing protocols (“this new struct conforms to protocols X, Y, and Z”)	(weak point) we prefer to build functionality through class inheritance. However, OOP developers also usually prefer composing functionality to inheriting it => easier to maintain.
developers lean heavily on the protocol extension functionality of Swift to build types that get a lot of their behavior from protocols.	
=> Which easier to share functionality across many types	
=> Reduce code in big software project	