Protocols and Extensions

Use `protocol` to declare a protocol.

```swift
protocol ExampleProtocol {
    var simpleDescription: String { get }
    mutating func adjust()
}
```

Classes, enumerations, and structs can all adopt protocols.

```swift
class SimpleClass: ExampleProtocol {
    var simpleDescription: String = "A very simple class."
    var anotherProperty: Int = 69105
    func adjust() {
        simpleDescription += "  Now 100% adjusted."
    }
}
var a = SimpleClass()
a.adjust()
let aDescription = a.simpleDescription

struct SimpleStructure: ExampleProtocol {
    var simpleDescription: String = "A simple structure"
    mutating func adjust() {
        simpleDescription += " (adjusted)"
    }
}
var b = SimpleStructure()
b.adjust()
let bDescription = b.simpleDescription
```

EXPERIMENT

Add another requirement to `ExampleProtocol`. What changes do you need to make to `SimpleClass` and `SimpleStructure` so that they still conform to the protocol?

따르다,따르게하다

Notice the use of the `mutating` keyword in the declaration of `SimpleStructure` to mark a method that modifies the structure. The declaration of `SimpleClass` doesn't need any of its methods marked as mutating because methods on a class can always modify the class.

Use `extension` to add $\overset{\text{기능}}{\text{functionality}}$ to an existing type, such as new methods and computed properties. You can use an extension to add protocol $\overset{\text{적합성}}{\text{conformance}}$ to a type that's declared $\overset{\text{다른곳}}{\text{elsewhere}}$, or even to a type that you imported from a library or framework.

```swift
extension Int: ExampleProtocol {
    var simpleDescription: String {
        return "The number \(self)"
    }
    mutating func adjust() {
        self += 42
    }
}
print(7.simpleDescription)
// Prints "The number 7"
```

EXPERIMENT

Write an extension for the `Double` type that adds an `absoluteValue` property.

You can use a protocol name just like any other named type—for example, to create a collection of objects that have different types but that all conform to a single protocol. When you work with values whose type is a protocol type, methods outside the protocol definition aren't available.

```swift
let protocolValue: ExampleProtocol = a
print(protocolValue.simpleDescription)
// Prints "A very simple class.  Now 100% adjusted."
// print(protocolValue.anotherProperty)  // Uncomment to see the error
```

$\overset{\text{일지라도}}{\text{Even though}}$ the variable `protocolValue` has a runtime type of `SimpleClass`, the compiler treats it as the given type of `ExampleProtocol`. This means that you can't accidentally access methods or properties that the class $\overset{\text{구현하다}}{\text{implements}}$ in addition to its protocol conformance.