Protocols

Swift

Protocols

A *protocol* defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality.

```
protocol SomeProtocol {
   // protocol definition goes here
}
struct SomeStructure: FirstProtocol, AnotherProtocol {
 // structure definition goes here
}
class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol {
   // class definition goes here
}
```

Property Requirements

A protocol can require any conforming type to provide an instance property or type property with a particular name and type.

```
protocol SomeProtocol {
    var mustBeSettable: Int { get set }
    var doesNotNeedToBeSettable: Int { get }
}
```

Here's an example of a protocol with a single instance property requirement:

```
protocol FullyNamed {
    var fullName: String { get }
}
```

```
struct Person: FullyNamed {
    var fullName: String
}
```

```
let john = Person(fullName: "John Appleseed")
// john.fullName is "John Appleseed"
```

Here's a more complex class, which also adopts and conforms to the `FullyNamed` protocol:

```
class Starship: FullyNamed {
    var prefix: String?
    var name: String
    init(name: String, prefix: String? = nil) {
        self.name = name
        self.prefix = prefix
    }
    var fullName: String {
        return (prefix != nil ? prefix! + " " : "") + name
    }
}
var ncc1701 = Starship(name: "Enterprise", prefix: "USS")
// ncc1701.fullName is "USS Enterprise"
```

Method requirements

Protocols can require specific instance methods and type methods to be implemented by conforming types.

```
protocol SomeProtocol {
    static func someTypeMethod()
}
protocol RandomNumberGenerator {
    func random() -> Double
}
```

Here's an implementation of a class that adopts and conforms to the `RandomNumberGenerator` protocol.

```
class LinearCongruentialGenerator: RandomNumberGenerator {
    var lastRandom = 42.0
    let m = 139968.0
    let a = 3877.0
    let c = 29573.0
    func random() -> Double {
        lastRandom = ((lastRandom * a + c).truncatingRemainder(dividingBy:m))
        return lastRandom / m
    }
}
let generator = LinearCongruentialGenerator()
print("Here's a random number: \(generator.random())")
// Prints "Here's a random number: 0.3746499199817l"
print("And another one: \(generator.random())")
// Prints "And another one: 0.729023776863283"
```

Mutating Method Requirements

It is sometimes necessary for a method to modify (or *mutate*) the instance it belongs to.

```
protocol Togglable {
    mutating func toggle()
}
```

If you define a protocol instance method requirement that is intended to mutate instances of any type that adopts the protocol, mark the method with the `mutating` keyword as part of the protocol's definition.

The example below defines an enumeration called `OnOffSwitch`.

```
enum OnOffSwitch: Togglable {
    case off, on
    mutating func toggle() {
        switch self {
        case .off:
            self = .on
        case .on:
            self = .off
        }
    }
}
var lightSwitch = OnOffSwitch.off
lightSwitch.toggle()
// lightSwitch is now equal to .on
```

Initializer Requirements

Protocols can require specific initializers to be implemented by conforming types.

```
protocol SomeProtocol {
    init(someParameter: Int)
}

class SomeClass: SomeProtocol {
    required init(someParameter: Int) {
        // initializer implementation goes here
    }
}
```

Protocols as Types

Because it is a type, you can use a protocol in many places where other types are allowed, including:

- As a parameter type or return type in a function, method, or initializer
- As the type of a constant, variable, or property
- As the type of items in an array, dictionary, or other container

Here's an example of a protocol used as a type:

```
class Dice {
    let sides: Int
    let generator: RandomNumberGenerator
    init(sides: Int, generator: RandomNumberGenerator) {
        self.sides = sides
        self.generator = generator
    }
    func roll() -> Int {
        return Int(generator.random() * Double(sides)) + 1
    }
}
```

Here's how the `Dice` class can be used to create a six-sided dice with a `LinearCongruentialGenerator` instance as its random number generator:

```
var d6 = Dice(sides: 6, generator: LinearCongruentialGenerator())
for _ in 1...5 {
    print("Random dice roll is \(d6.roll())")
}
// Random dice roll is 3
// Random dice roll is 5
// Random dice roll is 4
// Random dice roll is 5
// Random dice roll is 4
```

Delegates

Delegation is a design pattern that enables a class or structure to hand off (or *delegate*) some of its responsibilities to an instance of another type.

```
protocol DiceGame {
    var dice: Dice { get }
    func play()
}
protocol DiceGameDelegate {
    func gameDidStart(_ game: DiceGame)
    func game(_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int)
    func gameDidEnd(_ game: DiceGame)
}
```

```
class SnakesAndLadders: DiceGame {
    let finalSquare = 25
    let dice = Dice(sides: 6, generator: LinearCongruentialGenerator())
    var square = 0
    var board: [Int]
    init() {
        board = Array(repeating: 0, count: finalSquare + 1)
        board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
        board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
    }
    var delegate: DiceGameDelegate?
    func play() {
        square = 0
        delegate?.gameDidStart(self)
        while square != finalSquare {
            let diceRoll = dice.roll()
            delegate?.game(self, didStartNewTurnWithDiceRoll: diceRoll)
            switch square + diceRoll {
            case finalSquare:
                break
            case let newSquare where newSquare > finalSquare:
                continue
            default:
                square += diceRoll
                square += board[square]
            }
        }
        delegate?.gameDidEnd(self)
}
```

Adding Protocol Conformance with an Extension

You can extend an existing type to adopt and conform to a new protocol, even if you do not have access to the source code for the existing type.

```
protocol TextRepresentable {
    var textualDescription: String { get }
}
```

The `Dice` class from earlier can be extended to adopt and conform to `TextRepresentable`:

```
extension Dice: TextRepresentable {
    var textualDescription: String {
        return "A \(sides)-sided dice"
    }
}
```

Any `Dice` instance can now be treated as `TextRepresentable`:

```
let d12 = Dice(sides: 12, generator:
   LinearCongruentialGenerator())
print(d12.textualDescription)
// Prints "A 12-sided dice"
```

Declaring Protocol Adoption with an Extension

If a type already conforms to all of the requirements of a protocol, but has not yet stated that it adopts that protocol, you can make it adopt the protocol with an empty extension:

```
struct Hamster {
    var name: String
    var textualDescription: String {
        return "A hamster named \(name)"
    }
}
extension Hamster: TextRepresentable {}
```

Collections of Protocol Types

A protocol can be used as the type to be stored in a collection such as an array or a dictionary, as mentioned in Protocols as Types.

```
let things: [TextRepresentable] = [d12, simonTheHamster]
```

It is now possible to iterate over the items in the array, and print each item's textual description:

```
for thing in things {
    print(thing.textualDescription)
}
// A 12-sided dice
// A hamster named Simon
```

Protocol Inheritance

A protocol can *inherit* one or more other protocols and can add further requirements on top of the requirements it inherits.

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

Class-Only Protocols

You can limit protocol adoption to class types (and not structures or enumerations) by adding the `AnyObject` protocol to a protocol's inheritance list.

```
protocol SomeClassOnlyProtocol: AnyObject,
  SomeInheritedProtocol {
    // class-only protocol definition goes
  here
}
```

Protocol Composition

It can be useful to require a type to conform to multiple protocols at once.

```
protocol Named {
    var name: String { get }
}
protocol Aged {
    var age: Int { get }
}
struct Person: Named, Aged {
    var name: String
    var age: Int
}
func wishHappyBirthday(to celebrator: Named & Aged) {
    print("Happy birthday, \(celebrator.name), you're \
   (celebrator.age)!")
}
let birthdayPerson = Person(name: "Malcolm", age: 21)
wishHappyBirthday(to: birthdayPerson)
// Prints "Happy birthday, Malcolm, you're 21!"
```

Checking for Protocol Conformance

You can use the `is` and `as` operators to check for protocol conformance, and to cast to a specific protocol.

This example defines a protocol called `HasArea`, with a single property requirement of a gettable `Double` property called `area`:

```
protocol HasArea {
    var area: Double { get }
}
```

Here are two classes, `Circle` and `Country`, both of which conform to the `HasArea` protocol:

```
class Circle: HasArea {
    let pi = 3.1415927
    var radius: Double
    var area: Double { return pi * radius * radius }
    init(radius: Double) { self.radius = radius }
}
class Country: HasArea {
    var area: Double
    init(area: Double) { self.area = area }
}
```

Here's a class called `Animal`, which does not conform to the `HasArea` protocol:

```
class Animal {
    var legs: Int
    init(legs: Int) { self.legs = legs }
}
```

So instances of all three types used to initialize an array that stores values of type `AnyObject`:

```
let objects: [AnyObject] = [
    Circle(radius: 2.0),
    Country(area: 243_610),
    Animal(legs: 4)
]
```

The `objects` array can now be iterated, and each object in the array can be checked to see if it conforms to the `HasArea` protocol:

```
for object in objects {
    if let objectWithArea = object as? HasArea {
        print("Area is \(objectWithArea.area)")
    } else {
        print("Something that doesn't have an area")
    }
}
// Area is 12.5663708
// Area is 243610.0
// Something that doesn't have an area
```

Optional Protocol Requirements

You can define *optional requirements* for protocols,
These requirements do not have to be implemented
by types that conform to the protocol.

```
protocol CounterDataSource {
 optional func increment(forCount count: Int) -> Int
 optional var fixedIncrement: Int { get }
}
```

Providing Default Implementation

You can use protocol extensions to provide a default implementation to any method or computed property requirement of that protocol.

```
protocol PrettyTextRepresentable: TextRepresentable {
}

extension PrettyTextRepresentable  {
    var prettyTextualDescription: String {
        return textualDescription
    }
}
```

Adding Constraints to Protocol Extension

When you define a protocol extension, you can specify constraints that conforming types must satisfy before the methods and properties of the extension are available.

```
extension Collection where Iterator.Element: TextRepresentable {
    var textualDescription: String {
        let itemsAsText = self.map { $0.textualDescription }
        return "[" + itemsAsText.joined(separator: ", ") + "]"
    }
}
```

Consider the `Hamster` structure from before, which conforms to the `TextRepresentable` protocol, and an array of `Hamster` values:

```
let murrayTheHamster = Hamster(name: "Murray")
let morganTheHamster = Hamster(name: "Morgan")
let mauriceTheHamster = Hamster(name: "Maurice")
let hamsters = [murrayTheHamster, morganTheHamster, mauriceTheHamster]
```

Because `Array` conforms to `Collection` and the array's elements conform to the `TextRepresentable` protocol, the array can use the `textualDescription` property to get a textual representation of its contents:

```
print(hamsters.textualDescription)
// Prints "[A hamster named Murray, A hamster named Morgan, A hamster
   named Maurice]"
```

https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Protocols.html