import UIKit  
//: # Protocols  
//: ### A protocol defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality  
//: ### The protocol can then be adopted by a class, structure, or enumeration to provide an actual implementation of those requirements  
//: ### Any type that satisfies the requirements of a protocol is said to conform to that protocol  
//: ## Protocol Syntax  
//: - Callout(Syntax):  
//:\* protocol SomeProtocol {  
//:\* // protocol definition goes here  
//:\* }  
//: ### Multiple protocols can be listed, and are separated by commas:  
//:\* struct SomeStructure: FirstProtocol, AnotherProtocol {  
//:\* // structure definition goes here  
//:\* }  
//: ### If a class has a superclass, list the superclass name before any protocols it adopts, followed by a comma:  
//: - Callout(Syntax):  
//: \* class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol {  
//: \* // class definition goes here  
//: \* }  
//: ## Property Requirements  
//: ### The protocol doesn’t specify whether the property should be a stored property or a computed property  
//: ### it only specifies the required property name and type  
//: ### The protocol also specifies whether each property must be gettable or gettable and settable  
//: ### Property requirements are always declared as variable properties -> var keyword  
//: ### Gettable and settable properties are indicated by writing { get set } after their type declaration  
//: ### gettable properties are indicated by writing { get }  
//: - Callout(Syntax):  
//:\* protocol SomeProtocol {  
//:\* var mustBeSettable: Int { get set }  
//:\* var doesNotNeedToBeSettable: Int { get }  
//:\* }  
//: ### Always prefix type property requirements with the static keyword  
//: - Callout(Syntax):  
//: protocol AnotherProtocol {  
//: static var someTypeProperty: Int { get set }  
//: }  
//: ### Example:  
protocol FullyNamed {  
 var fullName: String { get }  
}  
  
struct Person: FullyNamed {  
 var fullName: String  
}  
let john = Person(fullName: "John Appleseed")  
// john.fullName is "John Appleseed"  
//: ### Example 2:  
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  
//: ### they are like property requirements  
//: ### the methods is written in the same way as for normal instance and type methods, but without curly braces or a method body  
//: ### Example:  
protocol RandomNumberGenerator {  
 func random() -> Double  
}  
//: implementation of a class  
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: 0.3746499199817101"  
print("And another one: \(generator.random())")  
// Prints "And another one: 0.729023776863283"  
//: ## Mutating Method Requirements  
//: ### you place the mutating keyword before a method’s func keyword to indicate that the method is allowed to modify the instance it belongs to and any properties of that instance  
//: ### If you mark a protocol instance method requirement as mutating, you don’t need to write the mutating keyword when writing an implementation of that method for a class  
//: ### Example:  
protocol Togglable {  
 mutating func toggle()  
}  
  
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  
//:\* protocol SomeProtocol {  
//:\* init(someParameter: Int)  
//:\* }  
//: ### Class implementation:  
//: ### required keyword is a must  
//:\* class SomeClass: SomeProtocol {  
//:\* required init(someParameter: Int) {  
//:\* // initializer implementation goes here  
//:\* }  
//:\* }  
//: ### You don’t need to mark protocol initializer implementations with the required modifier on classes that are marked with the final modifier  
//: ### If a subclass overrides a designated initializer from a superclass and also implements a matching initializer requirement from a protocol:  
protocol SomeProtocol {  
 init()  
}  
  
class SomeSuperClass {  
 init() {  
 // initializer implementation goes here  
 }  
}  
  
class SomeSubClass: SomeSuperClass, SomeProtocol {  
 // "required" from SomeProtocol conformance; "override" from SomeSuperClass  
 required override init() {  
 // initializer implementation goes here  
 }  
}  
//: ## Protocols as Types  
//: ### 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  
 }  
}  
//: ### initializer has a parameter called generator, which is also of type RandomNumberGenerator. You can pass a value of any conforming type in to this parameter when initializing a new Dice instance  
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  
//: ## Adding Protocol Conformance with an Extension  
//: ### You can extend an existing type to adopt and conform to a new protocol  
protocol TextRepresentable {  
 var textualDescription: String { get }  
}  
// The Dice class 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  
struct Hamster {  
 var name: String  
 var textualDescription: String {  
 return "A hamster named \(name)"  
 }  
}  
extension Hamster: TextRepresentable {}  
  
let simonTheHamster = Hamster(name: "Simon")  
let somethingTextRepresentable: TextRepresentable = simonTheHamster  
print(somethingTextRepresentable.textualDescription)  
// Prints "A hamster named Simon"  
//: ## Protocol Extensions  
//: ### Protocols can be extended to provide method, initializer, subscript, and computed property implementations to conforming types  
//: ### Example:  
extension RandomNumberGenerator {  
 func randomBool() -> Bool {  
 return random() > 0.5  
 }  
}  
//: ### By creating an extension on the protocol, all conforming types automatically gain this method implementation without any additional modification.  
//let generator = LinearCongruentialGenerator()  
print("Here's a random number: \(generator.random())")  
// Prints "Here's a random number: 0.3746499199817101"  
print("And here's a random Boolean: \(generator.randomBool())")  
// Prints "And here's a random Boolean: true"  
  
let a = remainder(9.0, 8.0)  
let b = 9.0.truncatingRemainder(dividingBy: 6.0)  
  
protocol sample {  
 var test : Bool { get }  
}  
  
extension sample {  
 var test : Bool {  
 return true  
 }  
}