Sun*

Protocol

Outline

1. Definition

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

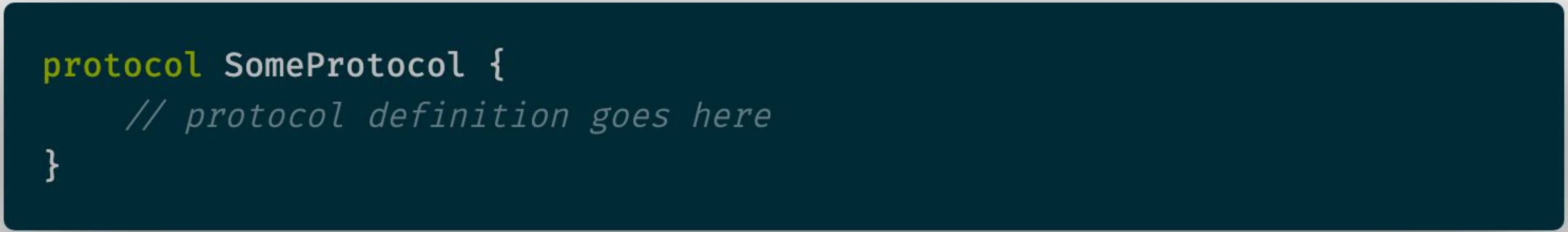
❖ The protocol can then be *adopted* by a class, structure, or enumeration to provide an actual implementation of those requirements. Any type that satisfies the requirements of a protocol is said to *conform* to that protocol.

2. Protocol Declarations

1. Protocol Syntax
2. Protocol Inheritance
3. Class-Only Protocols

2.1 Protocol Syntax

❖ You define protocols in a very similar way to classes, structures and enumerations:

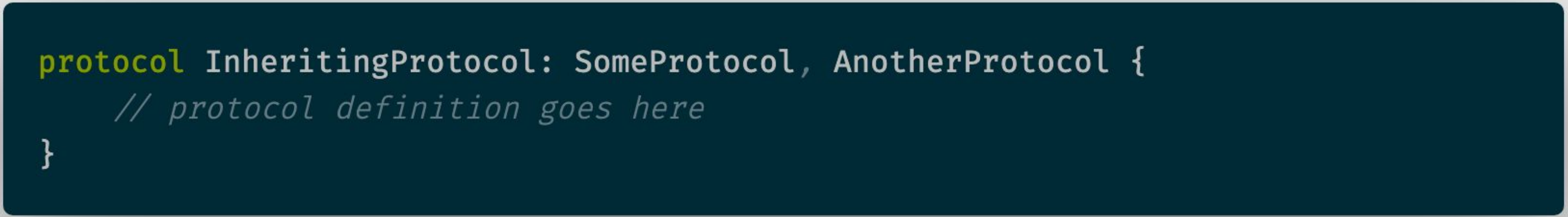


```
protocol SomeProtocol {
    // protocol definition goes here
}
```

2.2 Protocol Inheritance

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

❖ The syntax for protocol inheritance is similar to the syntax for class inheritance, but with the option to list multiple inherited protocols, separated by commas:

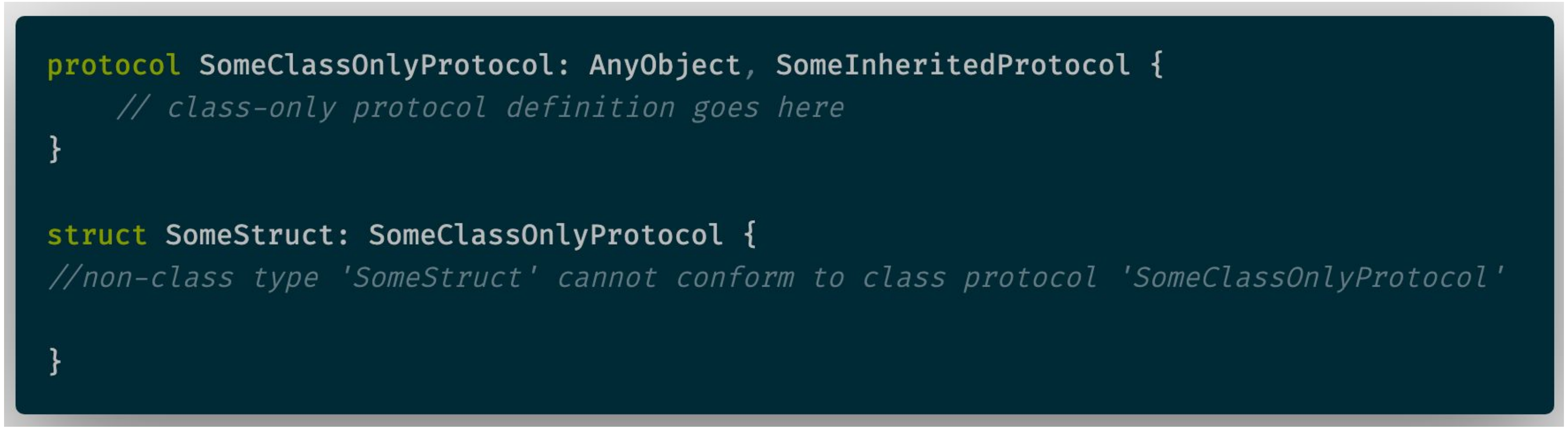


```
protocol InheritingProtocol: SomeProtocol, AnotherProtocol {
    // protocol definition goes here
}
```

Sun*

2.3 Class-Only Protocol

❖ 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
}

struct SomeStruct: SomeClassOnlyProtocol {
//non-class type 'SomeStruct' cannot conform to class protocol 'SomeClassOnlyProtocol'

}
```

3. Protocol Requirements

1. Property Requirements
2. Method Requirements
3. Mutating Method Requirements
4. Initializer Requirements
5. Optional Protocol Requirements

Sun*

3.1 Property Requirements

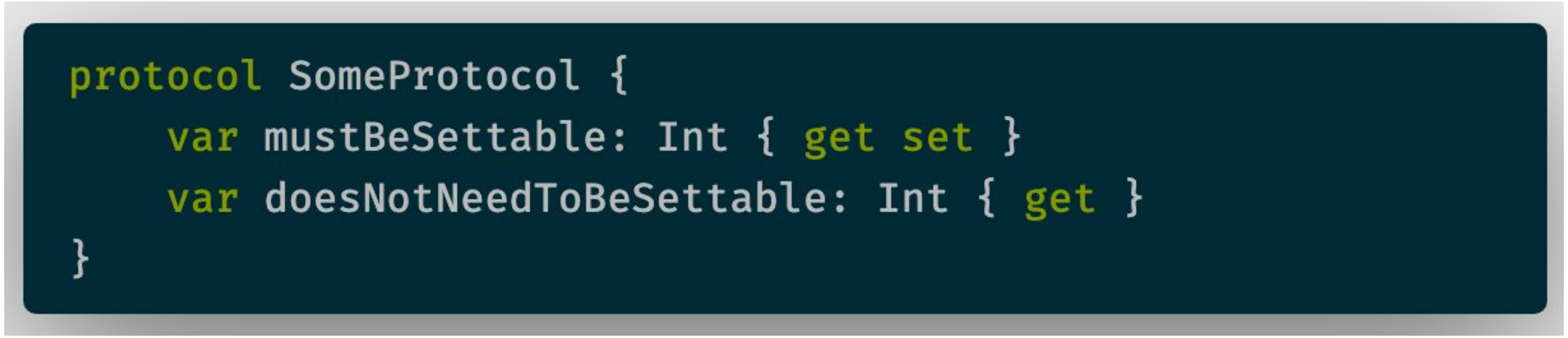
- ❖ A protocol can require any conforming type to provide an instance property or type property with a particular name and type. The protocol doesn’t specify whether the property should be a stored property or a computed property—it only specifies the required property name and type.

- ❖ The protocol also specifies whether each property must be gettable or gettable and settable.

Sun*

3.1 Property Requirements (cont)

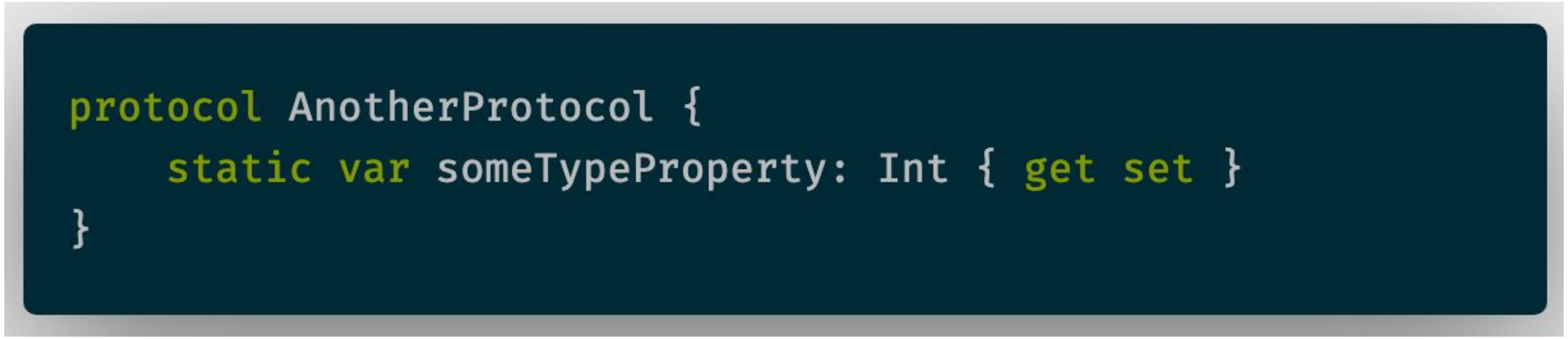
❖ Property requirements are always declared as variable properties, prefixed with the `var` keyword. Gettable and settable properties are indicated by writing `{ get set }` after their type declaration, and gettable properties are indicated by writing `{ get }`.



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

3.1 Property Requirements (cont)

- ❖ Always prefix type property requirements with the **static** keyword when you define them in a protocol. This rule pertains even though type property requirements can be prefixed with the **class** or **static** keyword when implemented by a class:



```
protocol AnotherProtocol {
    static var someTypeProperty: Int { get set }
}
```

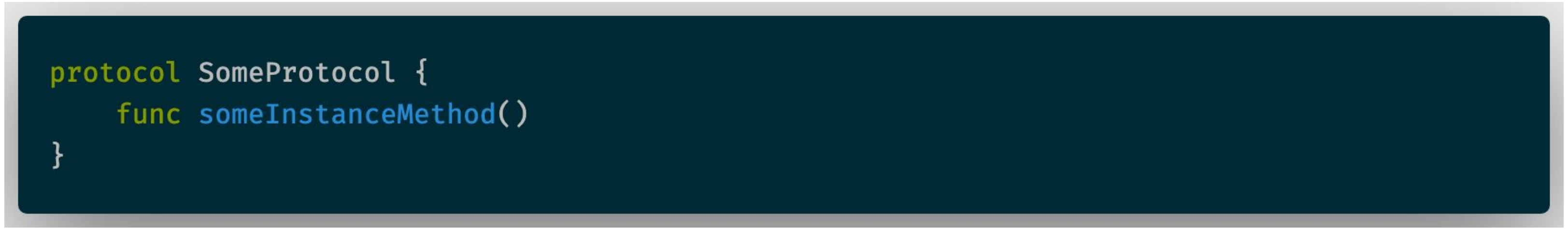
Sun*

3.2 Method Requirements

- ❖ Protocols can require specific instance methods and type methods to be implemented by conforming types.
- ❖ These methods are written as part of the protocol’s definition in exactly the same way as for normal instance and type methods, but without curly braces or a method body. Variadic parameters are allowed, subject to the same rules as for normal methods.
- ❖ Default values, however, can’t be specified for method parameters within a protocol’s definition.

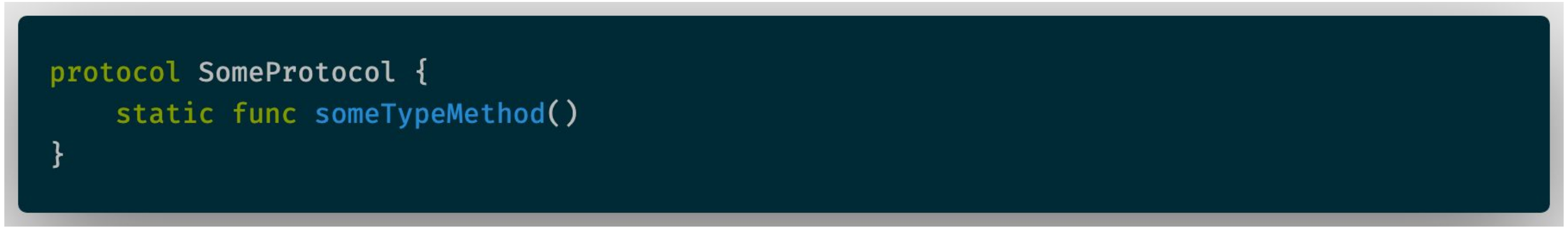
Sun*

3.2. Method Requirements (cont)



```
protocol SomeProtocol {
    func someInstanceMethod()
}
```

❖ As with type property requirements, you always prefix type method requirements with the **static** keyword when they're defined in a protocol. This is true even though type method requirements are prefixed with the **class** or **static** keyword when implemented by a class:



```
protocol SomeProtocol {
    static func someTypeMethod()
}
```

Sun*

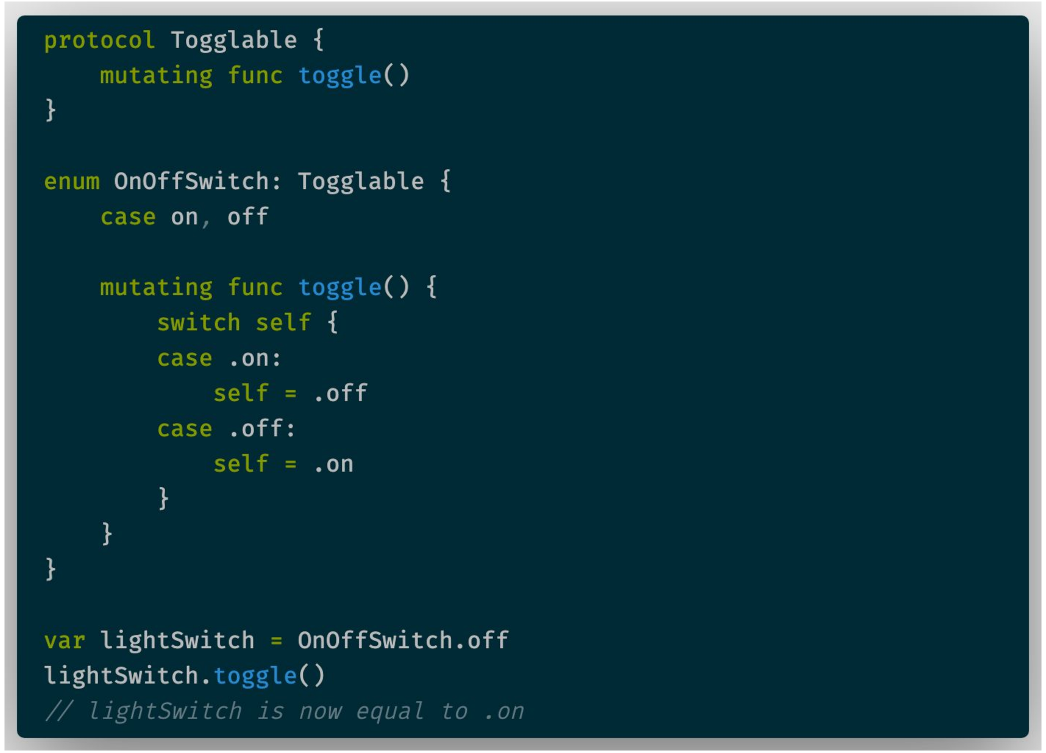
3.3 Mutating Method Requirements

❖ It’s sometimes necessary for a method to modify (or *mutate*) the instance it belongs to. For instance methods on value types (that is, structures and enumerations) 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.

Sun*

3.3 Mutating Method Requirements (cont)

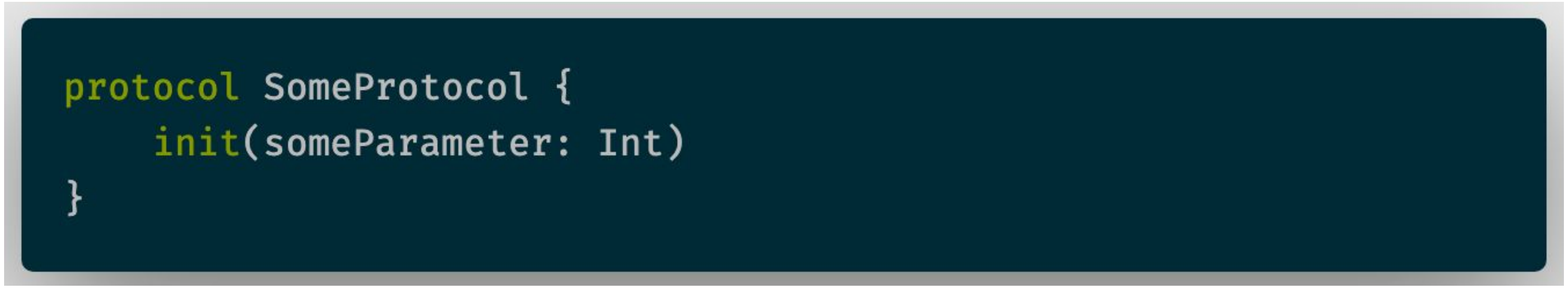
❖ Example:


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

enum OnOffSwitch: Togglable {
    case on, off

    mutating func toggle() {
        switch self {
        case .on:
            self = .off
        case .off:
            self = .on
        }
    }
}

var lightSwitch = OnOffSwitch.off
lightSwitch.toggle()
// lightSwitch is now equal to .on
```

Sun*

3.4 Initializer Requirements

❖ Protocols can require specific initializers to be implemented by conforming types. You write these initializers as part of the protocol's definition in exactly the same way as for normal initializers, but without curly braces or an initializer body:

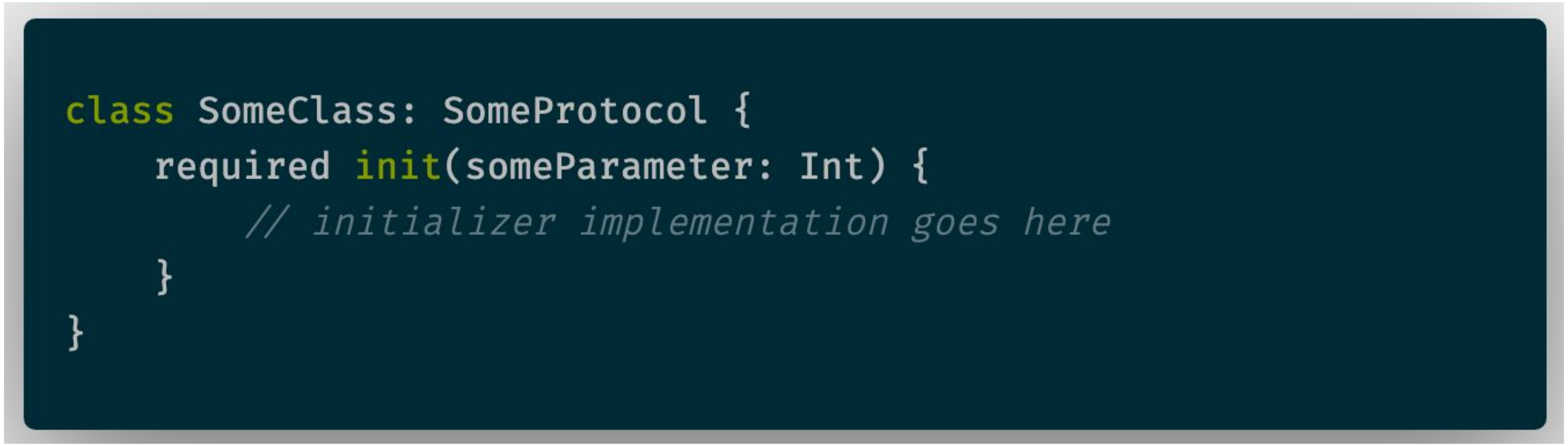


```
protocol SomeProtocol {
    init(someParameter: Int)
}
```

Sun*

3.4 Initializer Requirements (cont)

❖ You can implement a protocol initializer requirement on a conforming class as either a designated initializer or a convenience initializer. In both cases, you must mark the initializer implementation with the **`required`** modifier:



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

3.5 Optional Protocol Requirements

- ❖ You can define *optional requirements* for protocols. These requirements don't have to be implemented by types that conform to the protocol.

- ❖ Optional requirements are prefixed by the `optional` modifier as part of the protocol's definition. Optional requirements are available so that you can write code that interoperates with Objective-C. Both the protocol and the optional requirement must be marked with the `@objc` attribute.

- ❖ Note that `@objc` protocols can be adopted only by classes that inherit from Objective-C classes or other `@objc` classes. They can't be adopted by structures or enumerations.

4. Protocol Usage

1. Protocol as Types
2. Delegation
3. Adding Protocol Conformance with an Extension
4. Protocol Composition
5. Checking for Protocol Conformance
6. Protocol Extensions

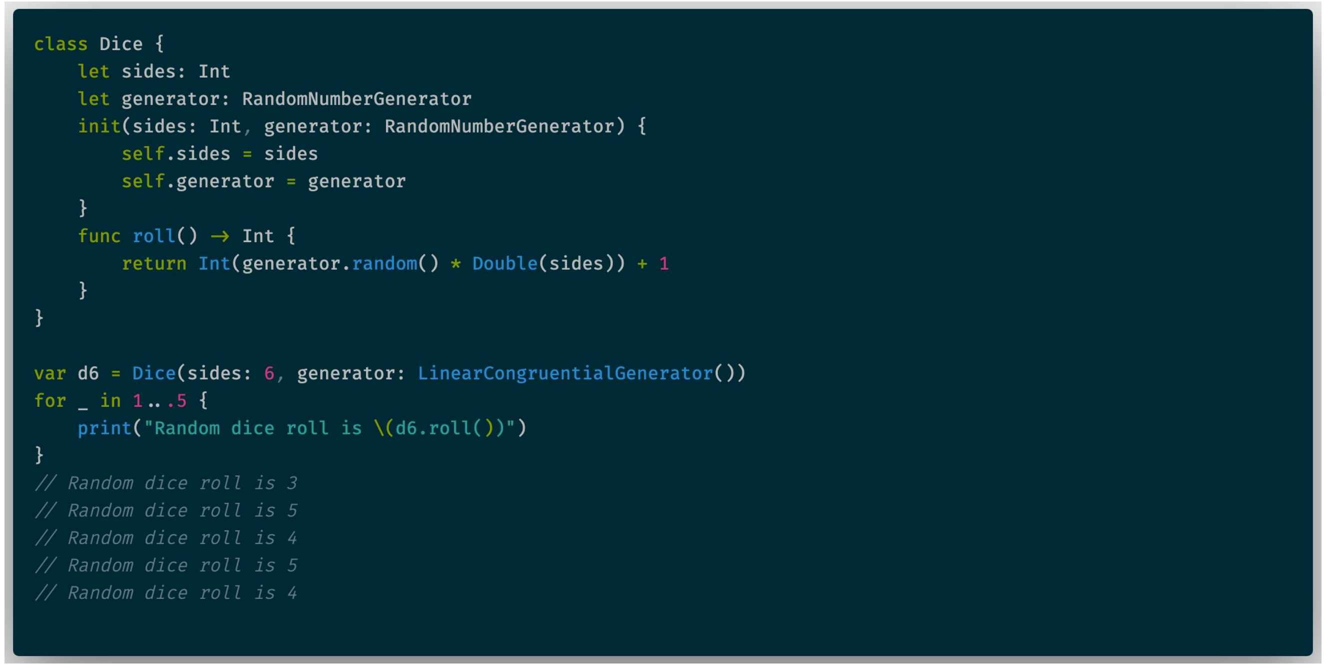
4.1 Protocol as Types

❖ Protocols don’t actually implement any functionality themselves. Nonetheless, you can use protocols as a fully fledged types in your code. Using a protocol as a type is sometimes called an *existential type*, which comes from the phrase “there exists a type *T* such that *T* conforms to the protocol”.

❖ 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

4.1 Protocol as Types (cont)

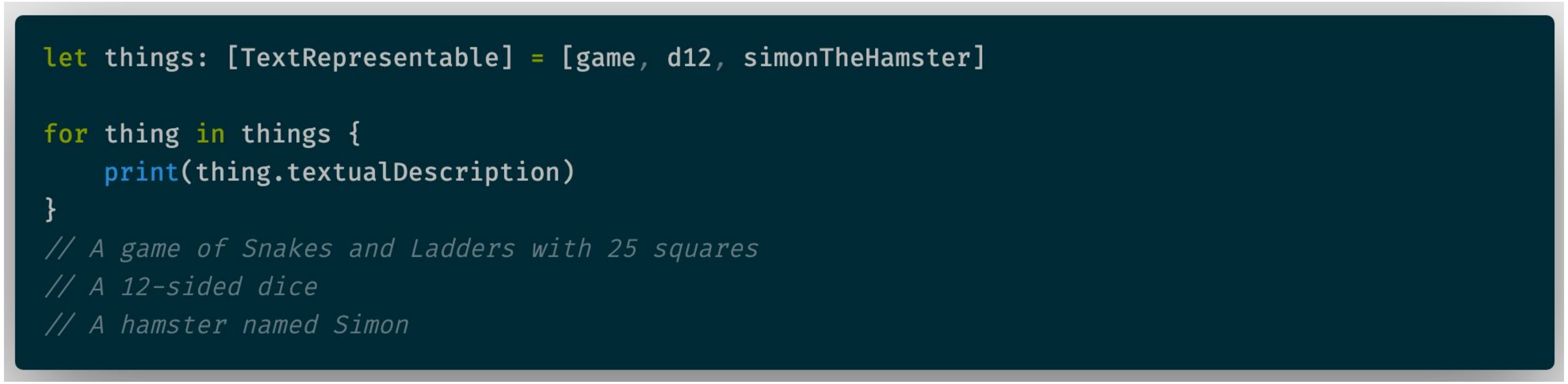
❖ Here’s an example of a protocol used as a type:



```swift
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
    }
}

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
```

4.1 Protocol as Types (cont)

- A protocol can be used as the type to be stored in a collection such as an array or a dictionary:



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

for thing in things {
    print(thing.textualDescription)
}
// A game of Snakes and Ladders with 25 squares
// A 12-sided dice
// A hamster named Simon
```

Sun*

4.2 Delegation

- *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. This design pattern is implemented by defining a protocol that encapsulates the delegated responsibilities, such that a conforming type (known as a delegate) is guaranteed to provide the functionality that has been delegated.

- Delegation can be used to respond to a particular action, or to retrieve data from an external source without needing to know the underlying type of that source.

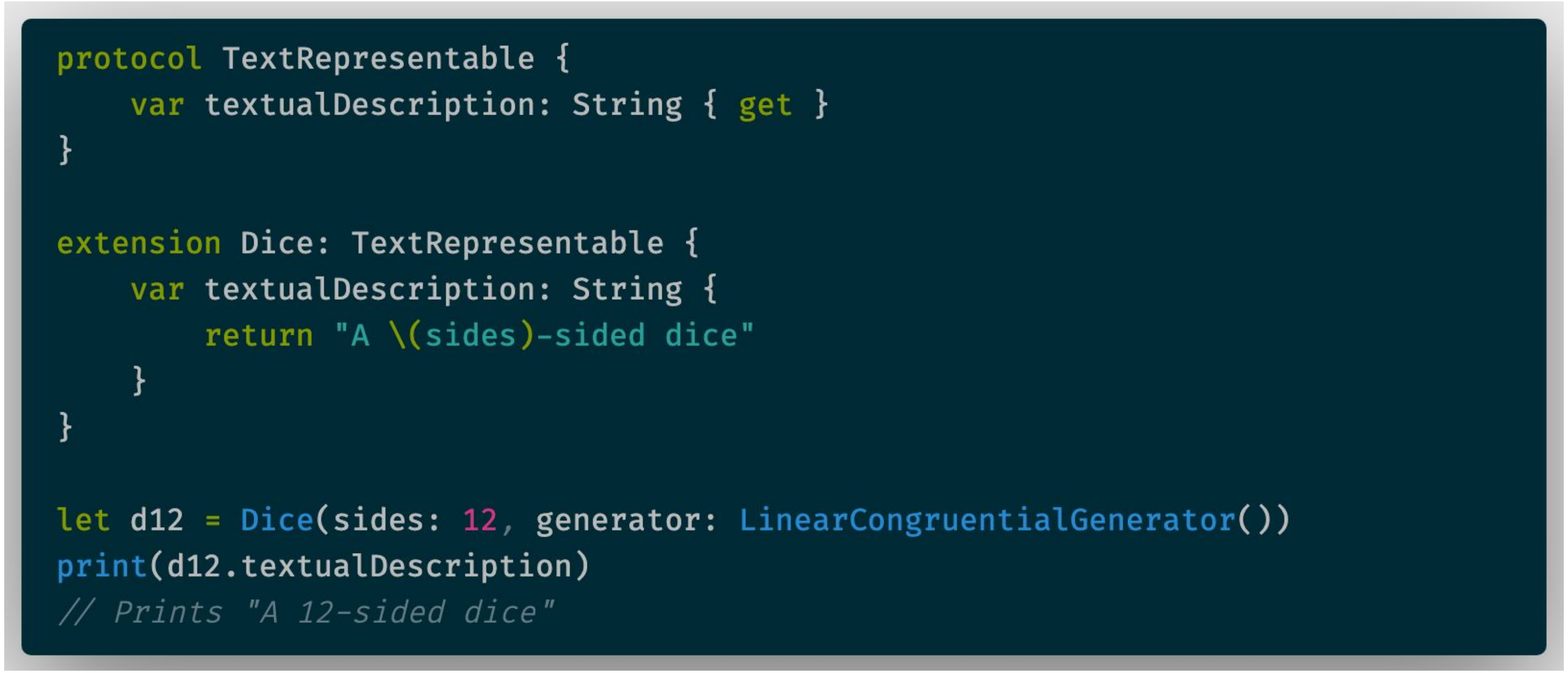
4.3 Adding Protocol Conformance with an Extension

- ❖ You can extend an existing type to adopt and conform to a new protocol, even if you don’t have access to the source code for the existing type.

- ❖ Extensions can add new properties, methods, and subscripts to an existing type, and are therefore able to add any requirements that a protocol may demand.

Sun*

4.3 Adding Protocol Conformance with an Extension

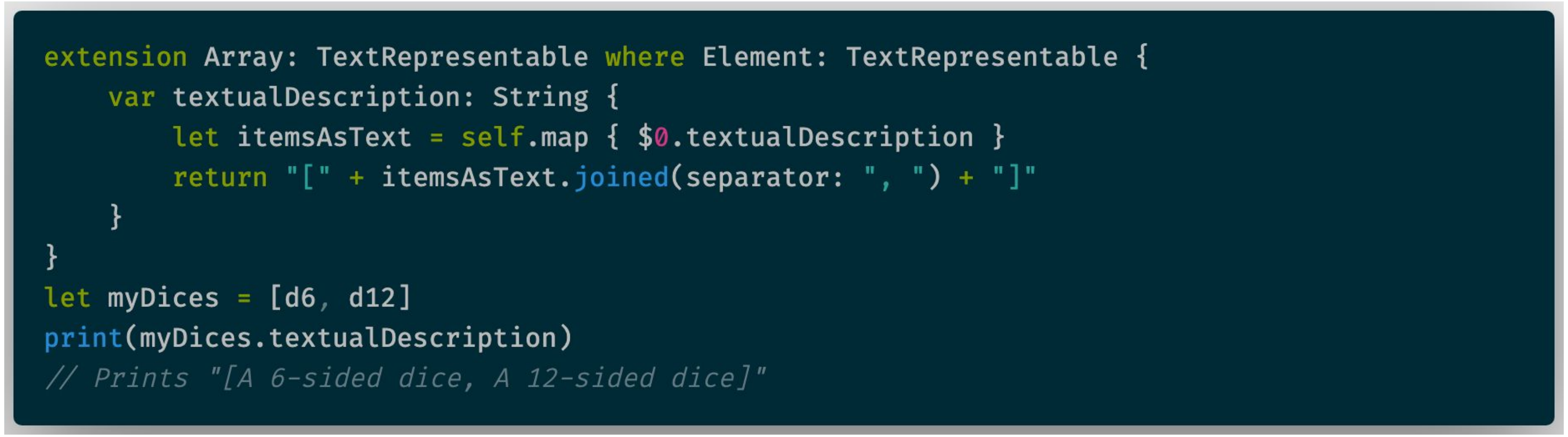


```swift
protocol TextRepresentable {
    var textualDescription: String { get }
}

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

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

Sun*

4.3 Adding Protocol Conformance with an Extension

❖ A generic type may be able to satisfy the requirements of a protocol only under certain conditions, such as when the type's generic parameter conforms to the protocol. You can make a generic type conditionally conform to a protocol by listing constraints when extending the type. Write these constraints after the name of the protocol you're adopting by writing a generic **`where`** clause.

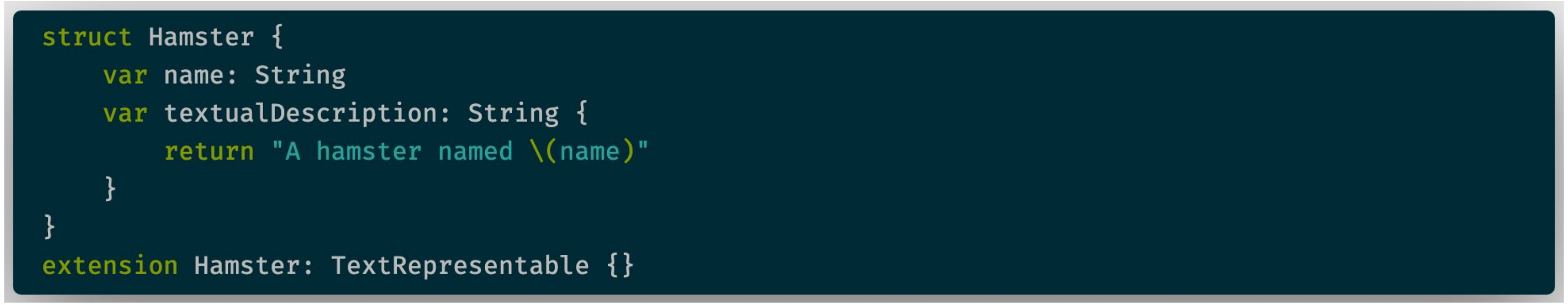


```swift
extension Array: TextRepresentable where Element: TextRepresentable {
    var textualDescription: String {
        let itemsAsText = self.map { $0.textualDescription }
        return "[" + itemsAsText.joined(separator: ", ") + "]"
    }
}
let myDices = [d6, d12]
print(myDices.textualDescription)
// Prints "[A 6-sided dice, A 12-sided dice]"
```

Sun*

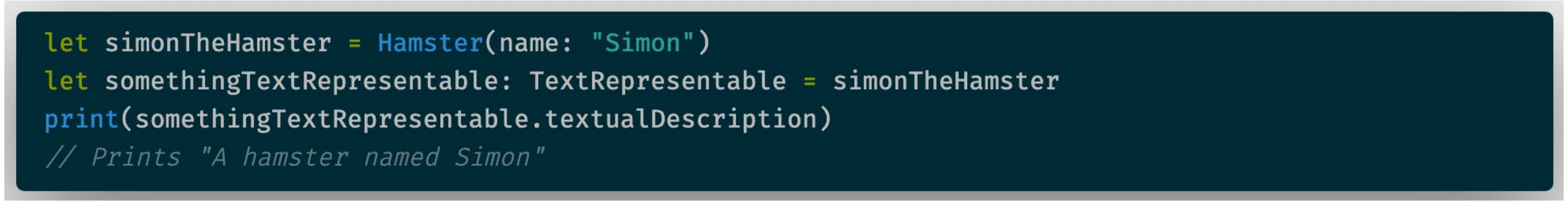
4.3 Adding Protocol Conformance with an Extension

- ❖ 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:



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

- ❖ Instances of **`Hamster`** can now be used wherever **`TextRepresentable`** is the required type:

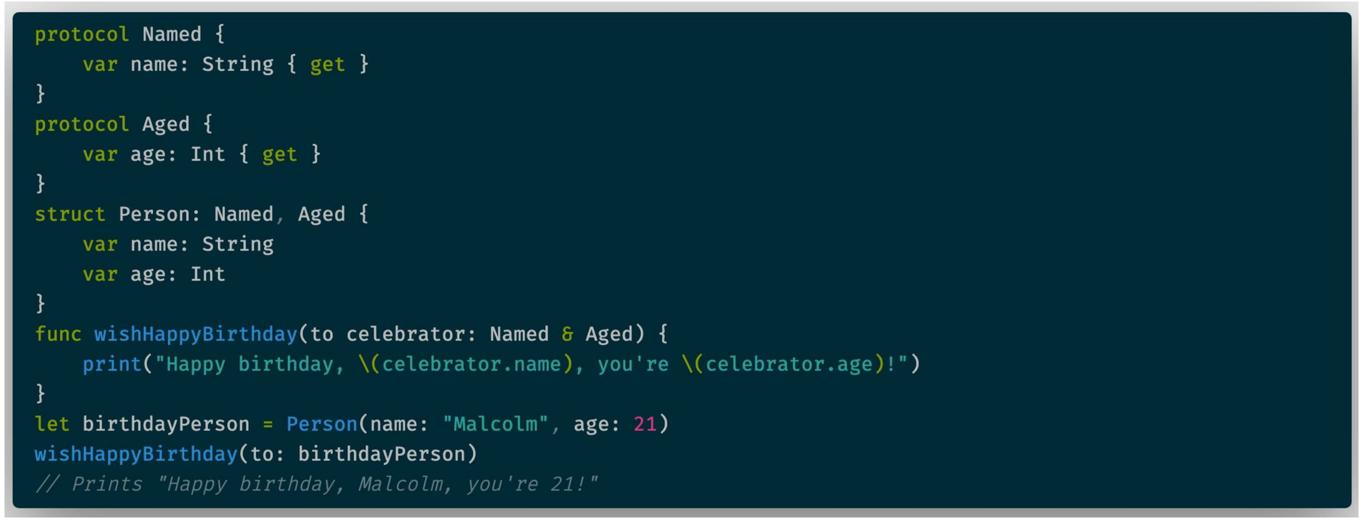


```swift
let simonTheHamster = Hamster(name: "Simon")
let somethingTextRepresentable: TextRepresentable = simonTheHamster
print(somethingTextRepresentable.textualDescription)
// Prints "A hamster named Simon"
```

Sun*

4.4 Protocol Composition

❖ It can be useful to require a type to conform to multiple protocols at the same time. You can combine multiple protocols into a single requirement with a *protocol composition*. Protocol compositions behave as if you defined a temporary local protocol that has the combined requirements of all protocols in the composition. Protocol compositions don’t define any new protocol types.



```swift
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!"
```

Sun*

4.5 Checking for Protocol Conformance

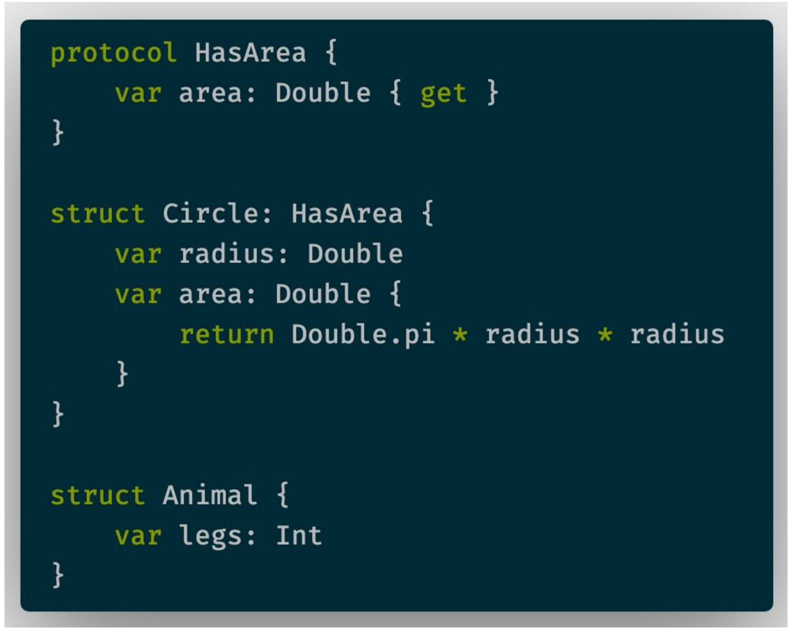
❖ You can use the `is` and `as` operators to check for protocol conformance and to cast to s specific protocol.

❖ Checking for and casting to a protocol follows exactly the same syntax as checking for and casting to a type:
  - ➔ The `is` operator returns `true` if an instance conforms to a protocol and returns `false` if it doesn’t.
  - ➔ The `as?` version of the downcast operator returns an optional value of the protocol’s type, and this value is `nil` if the instance doesn’t conform to that protocol.
  - ➔ The `as!` version of the downcast operator forces the downcast to the protocol type and triggers a runtime error if the downcast doesn’t succeed.

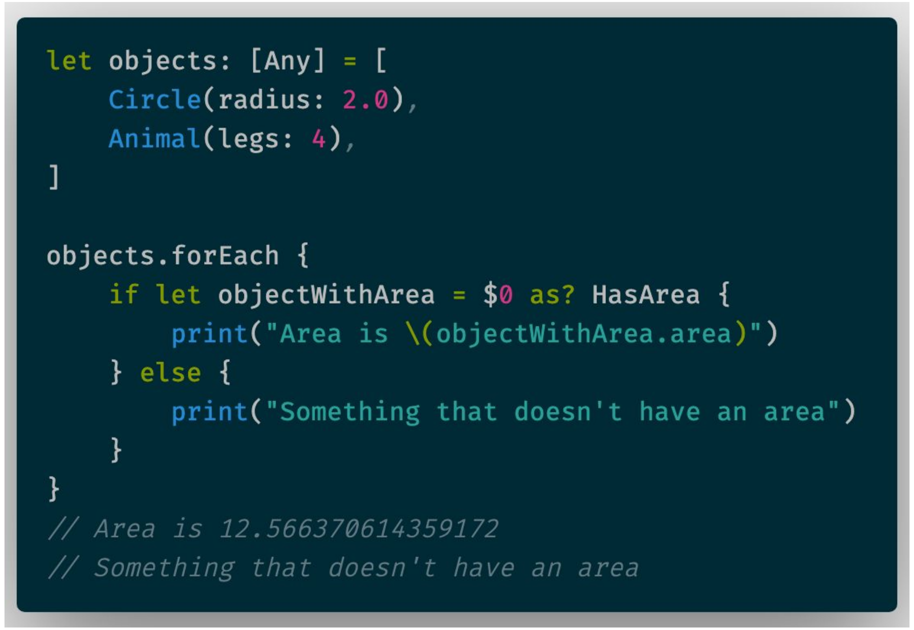
Sun*

4.5 Checking for Protocol Conformance (cont)

❖ Here is an example:



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

struct Circle: HasArea {
    var radius: Double
    var area: Double {
        return Double.pi * radius * radius
    }
}

struct Animal {
    var legs: Int
}
```

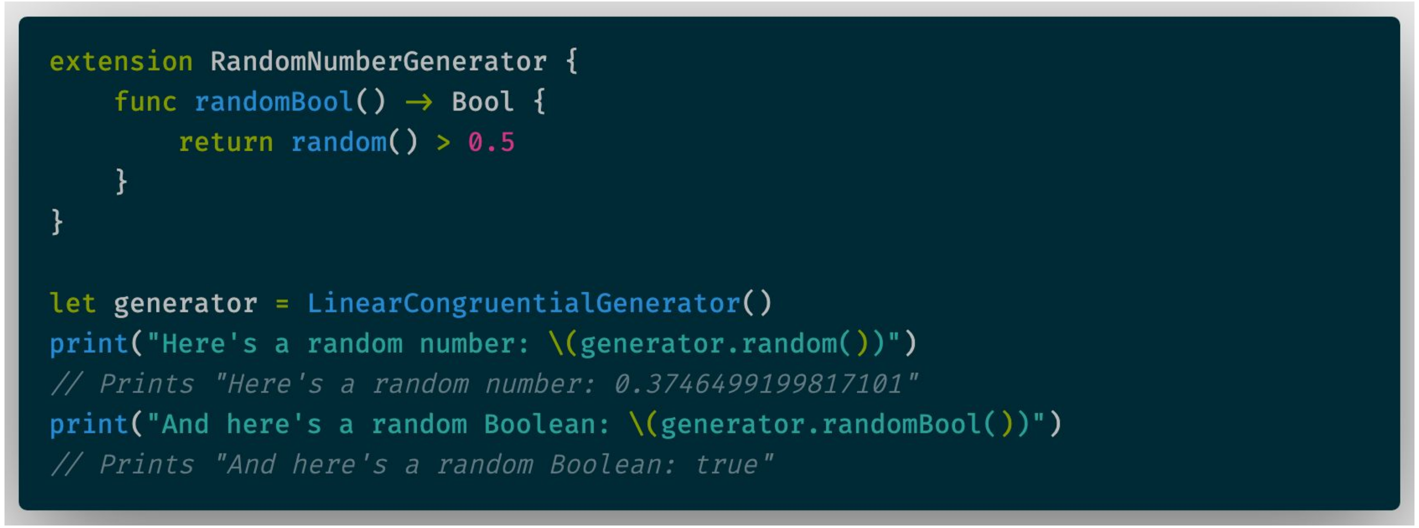


```swift
let objects: [Any] = [
    Circle(radius: 2.0),
    Animal(legs: 4),
]

objects.forEach {
    if let objectWithArea = $0 as? HasArea {
        print("Area is \(objectWithArea.area)")
    } else {
        print("Something that doesn't have an area")
    }
}
// Area is 12.566370614359172
// Something that doesn't have an area
```

Sun*

4.6 Protocol Extensions

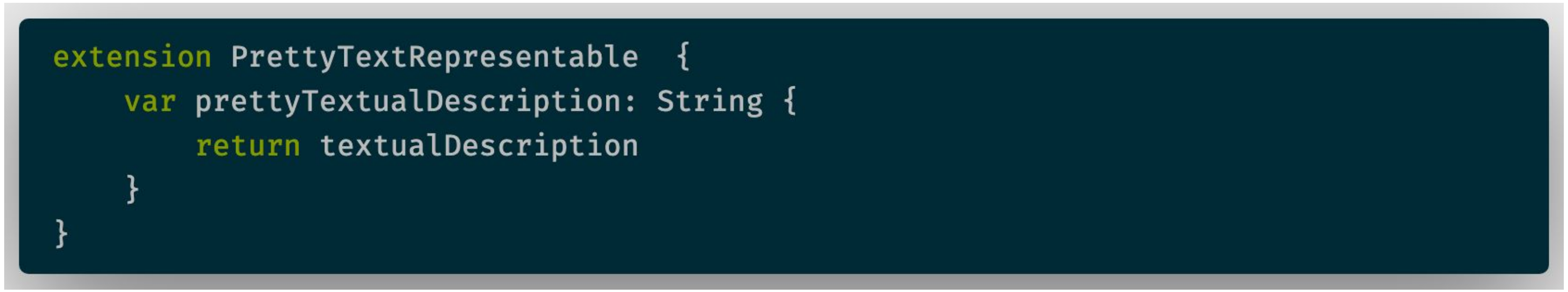
❖ Protocols can be extended to provide method, initializer, subscript, and computed property implementations to conforming types. This allows you to define behavior on protocols themselves, rather than in each type's individual conformance or in a global function.



```swift
extension RandomNumberGenerator {
    func randomBool() -> Bool {
        return random() > 0.5
    }
}

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"
```

4.6 Protocol Extensions (cont)

- You can use protocol extensions to provide a default implementation to any method or computed property requirement of that protocol. If a conforming type provides its own implementation of a required method or property, that implementation will be used instead of the one provided by the extension.

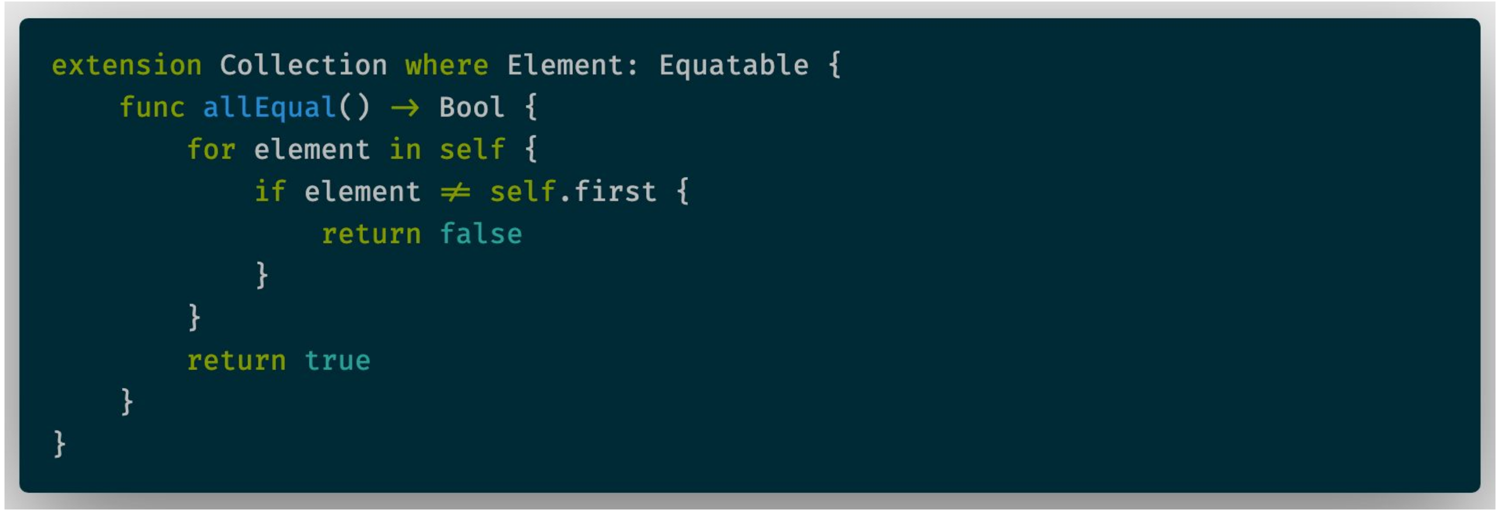


```swift
extension PrettyTextRepresentable  {
    var prettyTextualDescription: String {
        return textualDescription
    }
}
```

Sun*

4.6 Protocol Extensions (cont)

❖ 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. You write these constraints after the name of the protocol you're extending by writing a generic **`where`** clause.

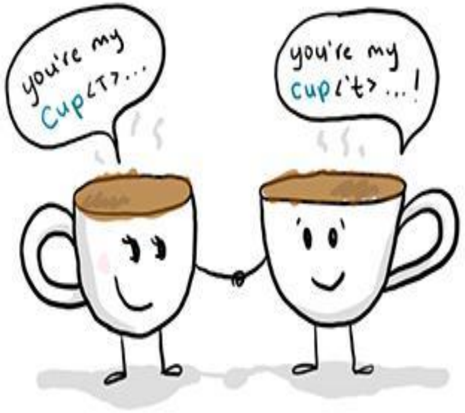


```
extension Collection where Element: Equatable {
    func allEqual() -> Bool {
        for element in self {
            if element != self.first {
                return false
            }
        }
        return true
    }
}
```

Sun*

Question & Answer?

THANK YOU


you're my cup<T>...
you're my cup<'t>...!