ON THIS PAGE

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.

In addition to specifying requirements that conforming types must implement, you can extend a protocol to implement some of these requirements or to implement additional functionality that conforming types can take advantage of.

Protocol Syntax

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

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

Custom types state that they adopt a particular protocol by placing the protocol's name after the type's name, separated by a colon, as part of their definition. 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:

```
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. 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.

If a protocol requires a property to be gettable and settable, that property requirement can't be fulfilled by a constant stored property or a read-only computed property. If the protocol only requires a property to be gettable, the requirement can be satisfied by any kind of property, and it's valid for the property to be also settable if this is useful for your own code.

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 }
}
```

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 }
```

```
}
```

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

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

The `FullyNamed` protocol requires a conforming type to provide a fully qualified name. The protocol doesn't specify anything else about the nature of the conforming type—it only specifies that the type must be able to provide a full name for itself. The protocol states that any `FullyNamed` type must have a gettable instance property called `fullName`, which is of type `String`.

Here's an example of a simple structure that adopts and conforms to the `FullyNamed` protocol:

```
struct Person: FullyNamed {
    var fullName: String
}
let john = Person(fullName: "John Appleseed")
// john.fullName is "John Appleseed"
```

This example defines a structure called `Person`, which represents a specific named person. It states that it adopts the `FullyNamed` protocol as part of the first line of its definition.

Each instance of `Person` has a single stored property called `fullName`, which is of type `String`. This matches the single requirement of the `FullyNamed` protocol, and means that `Person` has correctly conformed to the protocol. (Swift reports an error at compile-time if a protocol requirement is not fulfilled.)

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"
```

This class implements the `fullName` property requirement as a computed read-only property for a starship. Each `Starship` class instance stores a mandatory `name` and an optional `prefix`. The `fullName` property uses the `prefix` value if it exists, and prepends it to the beginning of `name` to create a full name for the starship.

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.

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()
}
```

The following example defines a protocol with a single instance method requirement:

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

This protocol, `RandomNumberGenerator`, requires any conforming type to have an instance method called `random`, which returns a `Double` value whenever it's called. Although it's not specified as part of the protocol, it's assumed that this value will be a number from `0.0` up to (but not including) `1.0`.

The `RandomNumberGenerator` protocol doesn't make any assumptions about how each random number will be generated—it simply requires the generator to provide a standard way to generate a new random number.

Here's an implementation of a class that adopts and conforms to the `RandomNumberGenerator` protocol. This class implements a pseudorandom number generator algorithm known as a *linear congruential generator*:

```
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

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. This process is described in Modifying Value Types from Within Instance Methods.

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. This enables structures and enumerations to adopt the protocol and satisfy that method requirement.

> NOTE
>
> 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. The `mutating` keyword is only used by structures and enumerations.

The example below defines a protocol called `Togglable`, which defines a single instance method requirement called `toggle`. As its name suggests, the `toggle()` method is intended to toggle or invert the state of any conforming type, typically by modifying a property of that type.

The `toggle()` method is marked with the `mutating` keyword as part of the `Togglable` protocol definition, to indicate that the method is expected to mutate the state of a conforming instance when it's called:

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

If you implement the `Togglable` protocol for a structure or enumeration, that structure or enumeration can conform to the protocol by providing an implementation of the `toggle()` method that is also marked as `mutating`.

The example below defines an enumeration called `OnOffSwitch`. This enumeration toggles between two states, indicated by the enumeration cases `on` and `off`. The enumeration's `toggle` implementation is marked as `mutating`, to match the `Togglable` protocol's requirements:

```
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. 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)
}
```

Class Implementations of Protocol Initializer Requirements

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
    }
}
```

The use of the `required` modifier ensures that you provide an explicit or inherited implementation of the initializer requirement on all subclasses of the conforming class, such that they also conform to the protocol.

For more information on required initializers, see Required Initializers.

> NOTE

> You don't need to mark protocol initializer implementations with the `required` modifier on classes that are marked with the `final` modifier, because final classes can't subclassed. For more about the `final` modifier, see Preventing Overrides.

If a subclass overrides a designated initializer from a superclass, and also implements a matching initializer requirement from a protocol, mark the initializer implementation with both the `required` and `override` modifiers:

```
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
    }
}
```

Failable Initializer Requirements

Protocols can define failable initializer requirements for conforming types, as defined in Failable Initializers.

A failable initializer requirement can be satisfied by a failable or nonfailable initializer on a conforming type. A nonfailable initializer requirement can be satisfied by a nonfailable initializer or an implicitly unwrapped failable initializer.

Protocols 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

> NOTE
>
> Because protocols are types, begin their names with a capital letter (such as `FullyNamed` and `RandomNumberGenerator`) to match the names of other types in Swift (such as `Int`, `String`, and `Double`).

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
    }
}
```

This example defines a new class called `Dice`, which represents an *n*-sided dice for use in a board game. `Dice` instances have an integer property called `sides`, which represents how many sides they have, and a property called `generator`, which provides a random number generator from which to create dice roll values.

The `generator` property is of type `RandomNumberGenerator`. Therefore, you can set it to an instance of *any* type that adopts the `RandomNumberGenerator` protocol. Nothing else is required of the instance you assign to this property, except that the instance must adopt the `RandomNumberGenerator` protocol. Because its type is `RandomNumberGenerator`, code inside the `Dice` class can only interact with `generator` in ways that apply to all generators that conform to this protocol. That means it can't use any methods or properties that are defined by the underlying type of the generator. However, you can downcast from a protocol type to an underlying type in the same way you can downcast from a superclass to a subclass, as discussed in Downcasting.

`Dice` also has an initializer, to set up its initial state. This 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.

`Dice` provides one instance method, `roll`, which returns an integer value between 1 and the number of sides on the dice. This method calls the generator's `random()` method to create a new random number between `0.0` and `1.0`, and uses this random number to create a dice roll value within the correct range. Because `generator` is known to adopt `RandomNumberGenerator`, it's guaranteed to have a `random()` method to call.

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
```

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.

The example below defines two protocols for use with dice-based board games:

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

The `DiceGame` protocol is a protocol that can be adopted by any game that involves dice.

The `DiceGameDelegate` protocol can be adopted to track the progress of a `DiceGame`. To prevent strong reference cycles, delegates are declared as weak references. For information about weak references, see Strong Reference Cycles Between Class Instances. Marking the protocol as class-only lets the `SnakesAndLadders` class later in this chapter declare that its delegate must use a weak reference. A class-only protocol is marked by its inheritance from `AnyObject` as discussed in Class-Only Protocols.

Here's a version of the *Snakes and Ladders* game originally introduced in Control Flow. This version is adapted to use a `Dice` instance for its dice-rolls; to adopt the `DiceGame` protocol; and to notify a `DiceGameDelegate` about its progress:

```
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
    }
    weak var delegate: DiceGameDelegate?
    func play() {
        square = 0
        delegate?.gameDidStart(self)
        gameLoop: while square != finalSquare {
            let diceRoll = dice.roll()
            delegate?.game(self,
  didStartNewTurnWithDiceRoll: diceRoll)
            switch square + diceRoll {
            case finalSquare:
                break gameLoop
            case let newSquare where newSquare >
  finalSquare:
                continue gameLoop
            default:
                square += diceRoll
                square += board[square]
            }
        }
        delegate?.gameDidEnd(self)
    }
}
```

For a description of the *Snakes and Ladders* gameplay, see Break.

This version of the game is wrapped up as a class called `SnakesAndLadders`, which adopts the `DiceGame` protocol. It provides a gettable `dice` property and a `play()` method in order to conform to the protocol. (The `dice` property is declared as a constant property because it doesn't need to change after initialization, and the protocol only requires that it must be gettable.)

The *Snakes and Ladders* game board setup takes place within the class's `init()` initializer. All game logic is moved into the protocol's `play` method, which uses the protocol's required `dice` property to provide its dice roll values.

Note that the `delegate` property is defined as an *optional* `DiceGameDelegate`, because a delegate isn't required in order to play the game. Because it's of an optional type, the `delegate` property is automatically set to an initial value of `nil`. Thereafter, the game instantiator has the option to set the property to a suitable delegate. Because the `DiceGameDelegate` protocol is class-only, you can declare the delegate to be `weak` to prevent reference cycles.

`DiceGameDelegate` provides three methods for tracking the progress of a game. These three methods have been incorporated into the game logic within the `play()` method above, and are called when a new game starts, a new turn begins, or the game ends.

Because the `delegate` property is an *optional* `DiceGameDelegate`, the `play()` method uses optional chaining each time it calls a method on the delegate. If the `delegate` property is nil, these delegate calls fail gracefully and without error. If the `delegate` property is non-nil, the delegate methods are called, and are passed the `SnakesAndLadders` instance as a parameter.

This next example shows a class called `DiceGameTracker`, which adopts the `DiceGameDelegate` protocol:

```
class DiceGameTracker: DiceGameDelegate {
    var numberOfTurns = 0
    func gameDidStart(_ game: DiceGame) {
        numberOfTurns = 0
        if game is SnakesAndLadders {
            print("Started a new game of Snakes and
  Ladders")
        }
        print("The game is using a \(game.dice.sides)-sided
```

```
    dice")
9       }
10      func game(_ game: DiceGame, didStartNewTurnWithDiceRoll
    diceRoll: Int) {
11          numberOfTurns += 1
12          print("Rolled a \(diceRoll)")
13      }
14      func gameDidEnd(_ game: DiceGame) {
15          print("The game lasted for \(numberOfTurns) turns")
16      }
17  }
```

`DiceGameTracker` implements all three methods required by `DiceGameDelegate`. It uses these methods to keep track of the number of turns a game has taken. It resets a `numberOfTurns` property to zero when the game starts, increments it each time a new turn begins, and prints out the total number of turns once the game has ended.

The implementation of `gameDidStart(_:)` shown above uses the `game` parameter to print some introductory information about the game that is about to be played. The `game` parameter has a type of `DiceGame`, not `SnakesAndLadders`, and so `gameDidStart(_:)` can access and use only methods and properties that are implemented as part of the `DiceGame` protocol. However, the method is still able to use type casting to query the type of the underlying instance. In this example, it checks whether `game` is actually an instance of `SnakesAndLadders` behind the scenes, and prints an appropriate message if so.

The `gameDidStart(_:)` method also accesses the `dice` property of the passed `game` parameter. Because `game` is known to conform to the `DiceGame` protocol, it's guaranteed to have a `dice` property, and so the `gameDidStart(_:)` method is able to access and print the dice's `sides` property, regardless of what kind of game is being played.

Here's how `DiceGameTracker` looks in action:

```
let tracker = DiceGameTracker()
let game = SnakesAndLadders()
game.delegate = tracker
game.play()
// Started a new game of Snakes and Ladders
```

```
// The game is using a 6-sided dice
// Rolled a 3
// Rolled a 5
// Rolled a 4
// Rolled a 5
// The game lasted for 4 turns
```

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. For more about extensions, see Extensions.

> NOTE
>
> Existing instances of a type automatically adopt and conform to a protocol when that conformance is added to the instance's type in an extension.

For example, this protocol, called `TextRepresentable`, can be implemented by any type that has a way to be represented as text. This might be a description of itself, or a text version of its current state:

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

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

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

This extension adopts the new protocol in exactly the same way as if `Dice` had provided it in its original implementation. The protocol name is provided after the type name, separated by a colon, and an implementation of all requirements of the protocol is provided within the extension's curly braces.

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

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

Similarly, the `SnakesAndLadders` game class can be extended to adopt and conform to the `TextRepresentable` protocol:

```
extension SnakesAndLadders: TextRepresentable {
    var textualDescription: String {
        return "A game of Snakes and Ladders with \
  (finalSquare) squares"
    }
}
print(game.textualDescription)
// Prints "A game of Snakes and Ladders with 25 squares"
```

Conditionally Conforming to a Protocol

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. For more about generic `where` clauses, see Generic Where Clauses.

The following extension makes `Array` instances conform to the `TextRepresentable` protocol whenever they store elements of a type that conforms to `TextRepresentable`.

```
extension Array: TextRepresentable where Element:
  TextRepresentable {
```

```
    var textualDescription: String {
        let itemsAsText = self.map { $0.textualDescription }
        return "[" + itemsAsText.joined(separator: ", ") + "]"
    }
}
let myDice = [d6, d12]
print(myDice.textualDescription)
// Prints "[A 6-sided dice, 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 {}
```

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

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

> NOTE
>
> Types don't automatically adopt a protocol just by satisfying its requirements. They must always explicitly declare their adoption of the protocol.

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. This example creates an array of `TextRepresentable` things:

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

It's 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 game of Snakes and Ladders with 25 squares
// A 12-sided dice
// A hamster named Simon
```

Note that the `thing` constant is of type `TextRepresentable`. It's not of type `Dice`, or `DiceGame`, or `Hamster`, even if the actual instance behind the scenes is of one of those types. Nonetheless, because it's of type `TextRepresentable`, and anything that is `TextRepresentable` is known to have a `textualDescription` property, it's safe to access `thing.textualDescription` each time through the loop.

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
}
```

Here's an example of a protocol that inherits the `TextRepresentable` protocol from above:

```
protocol PrettyTextRepresentable: TextRepresentable {
    var prettyTextualDescription: String { get }
}
```

This example defines a new protocol, `PrettyTextRepresentable`, which inherits from `TextRepresentable`. Anything that adopts `PrettyTextRepresentable` must satisfy all of the requirements enforced by `TextRepresentable`, *plus* the additional requirements enforced by `PrettyTextRepresentable`. In this example, `PrettyTextRepresentable` adds a single requirement to provide a gettable property called `prettyTextualDescription` that returns a `String`.

The `SnakesAndLadders` class can be extended to adopt and conform to `PrettyTextRepresentable`:

```
extension SnakesAndLadders: PrettyTextRepresentable {
    var prettyTextualDescription: String {
        var output = textualDescription + ":\n"
        for index in 1...finalSquare {
            switch board[index] {
            case let ladder where ladder > 0:
                output += "▲ "
            case let snake where snake < 0:
                output += "▼ "
            default:
                output += "○ "
            }
        }
        return output
    }
}
```

This extension states that it adopts the `PrettyTextRepresentable` protocol and provides an implementation of the `prettyTextualDescription` property for the `SnakesAndLadders` type. Anything that is `PrettyTextRepresentable` must also be `TextRepresentable`, and so the implementation of `prettyTextualDescription` starts by accessing the `textualDescription` property from the `TextRepresentable` protocol to begin an output string. It appends a colon and a line break, and uses this as the start of its pretty text representation. It then iterates through the array of board squares, and appends a geometric shape to represent the contents of each square:

- If the square's value is greater than `0`, it's the base of a ladder, and is represented by ▲.
- If the square's value is less than `0`, it's the head of a snake, and is represented by ▼.
- Otherwise, the square's value is `0`, and it's a "free" square, represented by ○.

The `prettyTextualDescription` property can now be used to print a pretty text description of any `SnakesAndLadders` instance:

```
print(game.prettyTextualDescription)
// A game of Snakes and Ladders with 25 squares:
// ○ ○ ▲ ○ ○ ▲ ○ ○ ▲ ▲ ○ ○ ○ ▼ ○ ○ ○ ○ ▼ ○ ○ ▼ ○ ▼ ○
```

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
}
```

In the example above, `SomeClassOnlyProtocol` can only be adopted by class types. It's a compile-time error to write a structure or enumeration definition that tries to adopt `SomeClassOnlyProtocol`.

> NOTE

> Use a class-only protocol when the behavior defined by that protocol's requirements assumes or requires that a conforming type has reference semantics rather than value semantics. For more about reference and value semantics, see Structures and Enumerations Are Value Types and Classes Are Reference Types.

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.

Protocol compositions have the form `SomeProtocol & AnotherProtocol`. You can list as many protocols as you need, separating them with ampersands (`&`). In addition to its list of protocols, a protocol composition can also contain one class type, which you can use to specify a required superclass.

Here's an example that combines two protocols called `Named` and `Aged` into a single protocol composition requirement on a function parameter:

```
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!"
```

In this example, the `Named` protocol has a single requirement for a gettable `String` property called `name`. The `Aged` protocol has a single requirement for a gettable `Int` property called `age`. Both protocols are adopted by a structure called `Person`.

The example also defines a `wishHappyBirthday(to:)` function. The type of the `celebrator` parameter is `Named & Aged`, which means "any type that conforms to both the `Named` and `Aged` protocols." It doesn't matter which specific type is passed to the function, as long as it conforms to both of the required protocols.

The example then creates a new `Person` instance called `birthdayPerson` and passes this new instance to the `wishHappyBirthday(to:)` function. Because `Person` conforms to both protocols, this call is valid, and the `wishHappyBirthday(to:)` function can print its birthday greeting.

Here's an example that combines the `Named` protocol from the previous example with a `Location` class:

```
class Location {
    var latitude: Double
    var longitude: Double
    init(latitude: Double, longitude: Double) {
        self.latitude = latitude
        self.longitude = longitude
    }
}
class City: Location, Named {
    var name: String
    init(name: String, latitude: Double, longitude: Double)
  {
        self.name = name
        super.init(latitude: latitude, longitude: longitude)
    }
}
func beginConcert(in location: Location & Named) {
    print("Hello, \(location.name)!")
}

let seattle = City(name: "Seattle", latitude: 47.6,
  longitude: -122.3)
beginConcert(in: seattle)
```

```
// Prints "Hello, Seattle!"
```

The `beginConcert(in:)` function takes a parameter of type `Location & Named`, which means "any type that's a subclass of `Location` and that conforms to the `Named` protocol." In this case, `City` satisfies both requirements.

Passing `birthdayPerson` to the `beginConcert(in:)` function is invalid because `Person` isn't a subclass of `Location`. Likewise, if you made a subclass of `Location` that didn't conform to the `Named` protocol, calling `beginConcert(in:)` with an instance of that type is also invalid.

Checking for Protocol Conformance

You can use the `is` and `as` operators described in Type Casting to check for protocol conformance, and to cast to a 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.

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 }
}
```

The `Circle` class implements the `area` property requirement as a computed property, based on a stored `radius` property. The `Country` class implements the `area` requirement directly as a stored property. Both classes correctly conform to the `HasArea` protocol.

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

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

The `Circle`, `Country` and `Animal` classes don't have a shared base class. Nonetheless, they're all classes, and so instances of all three types can be 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 is initialized with an array literal containing a `Circle` instance with a radius of 2 units; a `Country` instance initialized with the surface area of the United Kingdom in square kilometers; and an `Animal` instance with four legs.

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
```

Whenever an object in the array conforms to the `HasArea` protocol, the optional value returned by the `as?` operator is unwrapped with optional binding into a constant called `objectWithArea`. The `objectWithArea` constant is known to be of type `HasArea`, and so its `area` property can be accessed and printed in a type-safe way.

Note that the underlying objects aren't changed by the casting process. They continue to be a `Circle`, a `Country` and an `Animal`. However, at the point that they're stored in the `objectWithArea` constant, they're only known to be of type `HasArea`, and so only their `area` property can be accessed.

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.

When you use a method or property in an optional requirement, its type automatically becomes an optional. For example, a method of type `(Int) -> String` becomes `((Int) -> String)?`. Note that the entire function type is wrapped in the optional, not the method's return value.

An optional protocol requirement can be called with optional chaining, to account for the possibility that the requirement was not implemented by a type that conforms to the protocol. You check for an implementation of an optional method by writing a question mark after the name of the method when it's called, such as `someOptionalMethod?(someArgument)`. For information on optional chaining, see Optional Chaining.

The following example defines an integer-counting class called `Counter`, which uses an external data source to provide its increment amount. This data source is defined by the `CounterDataSource` protocol, which has two optional requirements:

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

The `CounterDataSource` protocol defines an optional method requirement called `increment(forCount:)` and an optional property requirement called `fixedIncrement`. These requirements define two different ways for data sources to provide an appropriate increment amount for a `Counter` instance.

> NOTE
>
> Strictly speaking, you can write a custom class that conforms to `CounterDataSource` without implementing *either* protocol requirement. They're both optional, after all. Although technically allowed, this wouldn't make for a very good data source.

The `Counter` class, defined below, has an optional `dataSource` property of type `CounterDataSource?`:

```
class Counter {
    var count = 0
    var dataSource: CounterDataSource?
    func increment() {
        if let amount = dataSource?.increment?(forCount:
  count) {
            count += amount
        } else if let amount = dataSource?.fixedIncrement {
            count += amount
```

```
            }
        }
    }
```

The `Counter` class stores its current value in a variable property called `count`. The `Counter` class also defines a method called `increment`, which increments the `count` property every time the method is called.

The `increment()` method first tries to retrieve an increment amount by looking for an implementation of the `increment(forCount:)` method on its data source. The `increment()` method uses optional chaining to try to call `increment(forCount:)`, and passes the current `count` value as the method's single argument.

Note that *two* levels of optional chaining are at play here. First, it's possible that `dataSource` may be `nil`, and so `dataSource` has a question mark after its name to indicate that `increment(forCount:)` should be called only if `dataSource` isn't `nil`. Second, even if `dataSource` *does* exist, there's no guarantee that it implements `increment(forCount:)`, because it's an optional requirement. Here, the possibility that `increment(forCount:)` might not be implemented is also handled by optional chaining. The call to `increment(forCount:)` happens only if `increment(forCount:)` exists—that is, if it isn't `nil`. This is why `increment(forCount:)` is also written with a question mark after its name.

Because the call to `increment(forCount:)` can fail for either of these two reasons, the call returns an *optional* `Int` value. This is true even though `increment(forCount:)` is defined as returning a non-optional `Int` value in the definition of `CounterDataSource`. Even though there are two optional chaining operations, one after another, the result is still wrapped in a single optional. For more information about using multiple optional chaining operations, see Linking Multiple Levels of Chaining.

After calling `increment(forCount:)`, the optional `Int` that it returns is unwrapped into a constant called `amount`, using optional binding. If the optional `Int` does contain a value—that is, if the delegate and method both exist, and the method returned a value—the unwrapped `amount` is added onto the stored `count` property, and incrementation is complete.

If it's *not* possible to retrieve a value from the `increment(forCount:)` method—either because `dataSource` is nil, or because the data source doesn't implement `increment(forCount:)`—then the `increment()` method tries to retrieve a value from the data source's `fixedIncrement` property instead. The `fixedIncrement` property is also an optional requirement, so its value is an optional `Int` value, even though `fixedIncrement` is defined as a non-optional `Int` property as part of the `CounterDataSource` protocol definition.

Here's a simple `CounterDataSource` implementation where the data source returns a constant value of `3` every time it's queried. It does this by implementing the optional `fixedIncrement` property requirement:

```
class ThreeSource: NSObject, CounterDataSource {
    let fixedIncrement = 3
}
```

You can use an instance of `ThreeSource` as the data source for a new `Counter` instance:

```
var counter = Counter()
counter.dataSource = ThreeSource()
for _ in 1...4 {
    counter.increment()
    print(counter.count)
}
// 3
// 6
// 9
// 12
```

The code above creates a new `Counter` instance; sets its data source to be a new `ThreeSource` instance; and calls the counter's `increment()` method four times. As expected, the counter's `count` property increases by three each time `increment()` is called.

Here's a more complex data source called `TowardsZeroSource`, which makes a `Counter` instance count up or down towards zero from its current `count` value:

```
class TowardsZeroSource: NSObject, CounterDataSource {
    func increment(forCount count: Int) -> Int {
```

```
        if count == 0 {
            return 0
        } else if count < 0 {
            return 1
        } else {
            return -1
        }
    }
}
```

The `TowardsZeroSource` class implements the optional `increment(forCount:)` method from the `CounterDataSource` protocol and uses the `count` argument value to work out which direction to count in. If `count` is already zero, the method returns `0` to indicate that no further counting should take place.

You can use an instance of `TowardsZeroSource` with the existing `Counter` instance to count from `-4` to zero. Once the counter reaches zero, no more counting takes place:

```
counter.count = -4
counter.dataSource = TowardsZeroSource()
for _ in 1...5 {
    counter.increment()
    print(counter.count)
}
// -3
// -2
// -1
// 0
// 0
```

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.

For example, the `RandomNumberGenerator` protocol can be extended to provide a `randomBool()` method, which uses the result of the required `random()` method to return a random `Bool` value:

```
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"
```

Protocol extensions can add implementations to conforming types but can't make a protocol extend or inherit from another protocol. Protocol inheritance is always specified in the protocol declaration itself.

Providing Default Implementations

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.

> NOTE
>
> Protocol requirements with default implementations provided by extensions are distinct from optional protocol requirements. Although conforming types don't have to provide their own implementation of either, requirements with default implementations can be called without optional chaining.

For example, the `PrettyTextRepresentable` protocol, which inherits the `TextRepresentable` protocol can provide a default implementation of its required `prettyTextualDescription` property to simply return the result of accessing the `textualDescription` property:

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

Adding Constraints to Protocol Extensions

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. For more about generic `where` clauses, see Generic Where Clauses.

For example, you can define an extension to the `Collection` protocol that applies to any collection whose elements conform to the `Equatable` protocol. By constraining a collection's elements to the `Equatable` protocol, a part of the standard library, you can use the `==` and `!=` operators to check for equality and inequality between two elements.

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

The `allEqual()` method returns `true` only if all the elements in the collection are equal.

Consider two arrays of integers, one where all the elements are the same, and one where they aren't:

```
let equalNumbers = [100, 100, 100, 100, 100]
let differentNumbers = [100, 100, 200, 100, 200]
```

Because arrays conform to `Collection` and integers conform to `Equatable`, `equalNumbers` and `differentNumbers` can use the `allEqual()` method:

```
print(equalNumbers.allEqual())
// Prints "true"
print(differentNumbers.allEqual())
// Prints "false"
```

> NOTE
>
> If a conforming type satisfies the requirements for multiple constrained extensions that provide implementations for the same method or property, Swift uses the implementation corresponding to the most specialized constraints.

‹ Extensions Generics ›

> BETA SOFTWARE
>
> This documentation contains preliminary information about an API or technology in development. This information is subject to change, and software implemented according to this documentation should be tested with final operating system software.
>
> Learn more about using Apple's beta software