Chapter 17: Protocols

Protocols are a way of specifying how to use an object. They describe a set of properties and methods which a class, structure, or enum should provide, although protocols pose no restrictions on the implementation.

Section 17.1: Protocol Basics

About Protocols

A Protocol specifies initialisers, properties, functions, subscripts and associated types required of a Swift object type (class, struct or enum) conforming to the protocol. In some languages similar ideas for requirement specifications of subsequent objects are known as 'interfaces'.

A declared and defined Protocol is a Type, in and of itself, with a signature of its stated requirements, somewhat similar to the manner in which Swift Functions are a Type based on their signature of parameters and returns.

Swift Protocol specifications can be optional, explicitly required and/or given default implementations via a facility known as Protocol Extensions. A Swift Object Type (class, struct or enum) desiring to conform to a Protocol that's fleshed out with Extensions for all its specified requirements needs only state its desire to conform to be in full conformance. The default implementations facility of Protocol Extensions can suffice to fulfil all obligations of conforming to a Protocol.

Protocols can be inherited by other Protocols. This, in conjunction with Protocol Extensions, means Protocols can and should be thought of as a significant feature of Swift.

Protocols and Extensions are important to realising Swift's broader objectives and approaches to program design flexibility and development processes. The primary stated purpose of Swift's Protocol and Extension capability is facilitation of compositional design in program architecture and development. This is referred to as Protocol Oriented Programming. Crusty old timers consider this superior to a focus on OOP design.

Protocols define interfaces which can be implemented by any struct, class, or enum:

```
protocol MyProtocol {
    init(value: Int)                    // required initializer
    func doSomething() -> Bool          // instance method
    var message: String { get }         // instance read-only property
    var value: Int { get set }          // read-write instance property
    subscript(index: Int) -> Int { get }  // instance subscript
    static func instructions() -> String  // static method
    static var max: Int { get }         // static read-only property
    static var total: Int { get set }     // read-write static property
}
```

Properties defined in protocols must either be annotated as `{ get }` or `{ get set }`. `{ get }` means that the property must be gettable, and therefore it can be implemented as *any* kind of property. `{ get set }` means that the property must be settable as well as gettable.

A struct, class, or enum may **conform to** a protocol:

```
struct MyStruct : MyProtocol {
    // Implement the protocol's requirements here
}
class MyClass : MyProtocol {
    // Implement the protocol's requirements here
}
```

```
enum MyEnum : MyProtocol {
    case caseA, caseB, caseC
    // Implement the protocol's requirements here
}
```

A protocol may also define a **default implementation** for any of its requirements through an extension:

```
extension MyProtocol {

    // default implementation of doSomething() -> Bool
    // conforming types will use this implementation if they don't define their own
    func doSomething() -> Bool {
        print("do something!")
        return true
    }
}
```

A protocol can be **used as a type**, provided it doesn't have `associatedtype` requirements:

```
func doStuff(object: MyProtocol) {
    // All of MyProtocol's requirements are available on the object
    print(object.message)
    print(object.doSomething())
}

let items : [MyProtocol] = [MyStruct(), MyClass(), MyEnum.caseA]
```

You may also define an abstract type that conforms to **multiple** protocols:

`Version ≥ 3.0`

With Swift 3 or better, this is done by separating the list of protocols with an ampersand (`&`):

```
func doStuff(object: MyProtocol & AnotherProtocol) {
    // ...
}

let items : [MyProtocol & AnotherProtocol] = [MyStruct(), MyClass(), MyEnum.caseA]
```

`Version < 3.0`

Older versions have syntax `protocol<...>` where the protocols are a comma-separated list between the angle brackets `<>`.

```
protocol AnotherProtocol {
    func doSomethingElse()
}

func doStuff(object: protocol<MyProtocol, AnotherProtocol>) {

    // All of MyProtocol & AnotherProtocol's requirements are available on the object
    print(object.message)
    object.doSomethingElse()
}

// MyStruct, MyClass & MyEnum must now conform to both MyProtocol & AnotherProtocol
let items : [protocol<MyProtocol, AnotherProtocol>] = [MyStruct(), MyClass(), MyEnum.caseA]
```

Existing types can be **extended** to conform to a protocol:

```
extension String : MyProtocol {
    // Implement any requirements which String doesn't already satisfy
}
```

Section 17.2: Delegate pattern

A *delegate* is a common design pattern used in Cocoa and CocoaTouch frameworks, where one class delegates responsibility for implementing some functionality to another. This follows a principle of separation of concerns, where the framework class implements generic functionality while a separate delegate instance implements the specific use case.

Another way to look into delegate pattern is in terms of object communication. `Objects` often needs to talks to each other and to do so an object need conform to a `protocol` in order to become a delegate of another Object. Once this setup has been done, the other object talks back to its delegates when interesting things happens.

For example, A view in userinterface to display a list of data should be responsible only for the logic of how data is displayed, not for deciding what data should be displayed.

Let's dive into a more concrete example. if you have two classes, a parent and a child:

```
class Parent { }
class Child { }
```

And you want to notify the parent of a change from the child.

In Swift, delegates are implemented using a `protocol` declaration and so we will declare a `protocol` which the `delegate` will implement. Here delegate is the `parent` object.

```
protocol ChildDelegate: class {
    func childDidSomething()
}
```

The child needs to declare a property to store the reference to the delegate:

```
class Child {
    weak var delegate: ChildDelegate?
}
```

Notice the variable `delegate` is an optional and the protocol `ChildDelegate` is marked to be only implemented by class type (without this the `delegate` variable can't be declared as a `weak` reference avoiding any retain cycle. This means that if the `delegate` variable is no longer referenced anywhere else, it will be released). This is so the parent class only registers the delegate when it is needed and available.

Also in order to mark our delegate as `weak` we must constrain our ChildDelegate protocol to reference types by adding `class` keyword in protocol declaration.

In this example, when the child does something and needs to notify its parent, the child will call:

```
delegate?.childDidSomething()
```

If the delegate has been defined, the delegate will be notified that the child has done something.

The parent class will need to extend the `ChildDelegate` protocol to be able to respond to its actions. This can be done directly on the parent class:

```
class Parent: ChildDelegate {
    ...

    func childDidSomething() {
        print("Yay!")
    }
}
```

Or using an extension:

```
extension Parent: ChildDelegate {
    func childDidSomething() {
        print("Yay!")
    }
}
```

The parent also needs to tell the child that it is the child's delegate:

```
// In the parent
let child = Child()
child.delegate = self
```

By default a Swift `protocol` does not allow an optional function be implemented. These can only be specified if your protocol is marked with the `@objc` attribute and the `optional` modifier.

For example `UITableView` implements the generic behavior of a table view in iOS, but the user must implement two delegate classes called `UITableViewDelegate` and `UITableViewDataSource` that implement how the specific cells look like and behave.

@objc public protocol UITableViewDelegate : NSObjectProtocol, UIScrollViewDelegate { // Display customization optional public func tableView(tableView: UITableView, willDisplayCell cell: UITableViewCell, forRowAtIndexPath indexPath: NSIndexPath) optional public func tableView(tableView: UITableView, willDisplayHeaderView view: UIView, forSection section: Int) optional public func tableView(tableView: UITableView, willDisplayFooterView view: UIView, forSection section: Int) optional public func tableView(tableView: UITableView, didEndDisplayingCell cell: UITableViewCell, forRowAtIndexPath indexPath: NSIndexPath) ... }

You can implement this protocol by changing your class definition, for example:

```
class MyViewController : UIViewController, UITableViewDelegate
```

Any methods not marked `optional` in the protocol definition (`UITableViewDelegate` in this case) must be implemented.

Section 17.3: Associated type requirements

Protocols may define **associated type requirements** using the `associatedtype` keyword:

```
protocol Container {
```

ciatedtype Element var count: Int { get } subscript(index: Int) -> Element { get set } }

Protocols with associated type requirements **can only be used as generic constraints**:

`// These are` allowed, because Container has associated type requirements: func displayValues(container: Container) { ... } class MyClass { let container: Container } // > error: protocol 'Container' can only be used as a

generic constraint // > because it has Self or associated type requirements // These are allowed: func displayValues**<T: Container>**(container: **T**) { ... } class MyClass**<T: Container>** { let container: **T** }

A type which conforms to the protocol may satisfy an `associatedtype` requirement implicitly, by providing a given type where the protocol expects the `associatedtype` to appear:

```
struct ContainerOfOne<T>: Container {
    let count = 1          // satisfy the count requirement
    var value: T

    // satisfy the subscript associatedtype requirement implicitly,
    // by defining the subscript assignment/return type as T
    // therefore Swift will infer that T == Element
    subscript(index: Int) ->
```

> { get { precondition(index == 0) return value } set { precondition(index == 0) value = newValue } } } let container = ContainerOfOne(value: "Hello")

(Note that to add clarity to this example, the generic placeholder type is named `T` – a more suitable name would be `Element`, which would shadow the protocol's `associatedtype Element`. The compiler will still infer that the generic placeholder `Element` is used to satisfy the `associatedtype Element` requirement.)

An `associatedtype` may also be satisfied explicitly through the use of a `typealias`:

```
struct ContainerOfOne<T>: Container {


```

alias Element = T subscript(index: Int) -> **Element** { ... } // ... }

The same goes for extensions:

```
// Expose an 8-bit integer as a collection of boolean values (one for each bit).
extension UInt8: Container {

    // as noted above, this typealias can be inferred
    typealias Element = Bool

    var count: Int { return 8 }
    subscript(index: Int) -> Bool {
        get {
            precondition(0 <= index && index < 8)
            return self & 1 << UInt8(index) != 0
        }
        set {
            precondition(0 <= index && index < 8)
            if newValue {
                self |= 1 << UInt8(index)
            } else {
                self &= ~(1 << UInt8(index))
            }
        }
    }
}
```

If the conforming type already satisfies the requirement, no implementation is needed:

```
extension Array: Container {}  // Array satisfies all requirements, including Element
```

Section 17.4: Class-Only Protocols

A protocol may specify that only a class can implement it through using the `class` keyword in its inheritance list. This keyword must appear before any other inherited protocols in this list.

`protocol ClassOnlyProtocol: s, SomeOtherProtocol { // Protocol requirements }`

If a non-class type tries to implement `ClassOnlyProtocol`, a compiler error will be generated.

```
struct MyStruct: ClassOnlyProtocol {
    // error: Non-class type 'MyStruct' cannot conform to class protocol 'ClassOnlyProtocol'
}
```

Other protocols may inherit from the `ClassOnlyProtocol`, but they will have the same class-only requirement.

```
protocol MyProtocol: ClassOnlyProtocol {
    // ClassOnlyProtocol Requirements
    // MyProtocol Requirements
}

class MySecondClass: MyProtocol {
    // ClassOnlyProtocol Requirements
    // MyProtocol Requirements
}
```

Reference semantics of class-only protocols

Using a class-only protocol allows for reference semantics when the conforming type is unknown.

```
protocol Foo : class {
    var bar : String { get set }
}

func takesAFoo(foo:Foo) {

    // this assignment requires reference semantics,
    // as foo is a let constant in this scope.
    foo.bar = "new value"
}
```

In this example, as `Foo` is a class-only protocol, the assignment to `bar` is valid as the compiler knows that `foo` is a class type, and therefore has reference semantics.

If `Foo` was not a class-only protocol, a compiler error would be yielded – as the conforming type could be a value type, which would require a `var` annotation in order to be mutable.

```
protocol Foo {
    var bar : String { get set }
}

func takesAFoo(foo:Foo) {
    foo.bar = "new value" // error: Cannot assign to property: 'foo' is a 'let' constant
}
```

```
func takesAFoo(foo:Foo) {
```

foo = foo // mutable copy of foo foo.bar = "new value" // no error – satisfies both reference and value semantics }

Weak variables of protocol type

When applying the weak modifier to a variable of protocol type, that protocol type must be class-only, as `weak` can only be applied to reference types.

```
weak var weakReference : ClassOnlyProtocol?
```

Section 17.5: Protocol extension for a specific conforming class

You can write the **default protocol implementation** for a specific class.

```
protocol MyProtocol {
    func doSomething()
}

extension MyProtocol where Self: UIViewController {
    func doSomething() {
        print("UIViewController default protocol implementation")
    }
}

class MyViewController: UIViewController, MyProtocol { }

let vc = MyViewController()
vc.doSomething() // Prints "UIViewController default protocol implementation"
```

Section 17.6: Using the RawRepresentable protocol (Extensible Enum)

```
// RawRepresentable has an associatedType RawValue.
// For this struct, we will make the compiler infer the type
// by implementing the rawValue variable with a type of String
//
// Compiler infers RawValue = String without needing typealias
//
struct NotificationName: RawRepresentable {
    let rawValue: String

    static let dataFinished = NotificationNames(rawValue: "DataFinishedNotification")
}
```

This struct can be extended elsewhere to add cases

```
extension NotificationName {
    static let documentationLaunched = NotificationNames(rawValue:
"DocumentationLaunchedNotification")
}
```

And an interface can design around any RawRepresentable type or specifically your enum struct

```
func post(notification notification: NotificationName) -> Void {
    // use notification.rawValue
}
```

At call site, you can use dot syntax shorthand for the typesafe NotificationName

```
post(notification: .dataFinished)
```

Using generic RawRepresentable function

```
// RawRepresentable has an associate type, so the
// method that wants to accept any type conforming to
// RawRepresentable needs to be generic
func observe<T: RawRepresentable>(object: T) -> Void {
    // object.rawValue
}
```

Section 17.7: Implementing Hashable protocol

Types used in `Sets` and `Dictionaries(key)` must conform to `Hashable` protocol which inherits from `Equatable` protocol.

Custom type conforming to `Hashable` protocol must implement

- A calculated property `hashValue`
- Define one of the equality operators i.e. `==` or `!=`.

Following example implements `Hashable` protocol for a custom `struct`:

```
struct Cell {
    var row: Int
    var col: Int

    init(_ row: Int, _ col: Int) {
        self.row = row
        self.col = col
    }
}

extension Cell: Hashable {

    // Satisfy Hashable requirement
    var hashValue: Int {
        get {
            return row.hashValue^col.hashValue
        }
    }

    // Satisfy Equatable requirement
    static func ==(lhs: Cell, rhs: Cell) -> Bool {
        return lhs.col == rhs.col && lhs.row == rhs.row
    }

}

// Now we can make Cell as key of dictonary
var dict = [Cell : String]()

dict[Cell(0, 0)] = "0, 0"
dict[Cell(1, 0)] = "1, 0"
dict[Cell(0, 1)] = "0, 1"

// Also we can create Set of Cells
```

```
var set = Set<Cell>()

set.insert(Cell(0, 0))
set.insert(Cell(1, 0))
```

Note: It is not necessary that different values in custom type have different hash values, collisions are acceptable. If hash values are equal, equality operator will be used to determine real equality.