Протоколы

Протокол - это перечисление свойств, методов и других требований. Они могут быть выполнены в классе, структуре или перечислениях. Говорят, что тип поддерживает(conform) протокол если он предоставляет реализацию всех требований. Обратите внимание на то, что в самом протоколе ничего не объявлено. Это просто список требований. Все свойства и методы должны быть реализованы непосредственно в типе, который поддерживает протокол.

Чтобы объявить новый протокол нужно использовать ключевое слово protocol. После него указывается имя, с большой буквы, и в фигурных скобках требования. Объявление очень похоже на структуры и классы.

```
// --- Протоколы CODE SNIPPET #1 --- //

protocol MyProtocol {

}
```

Давайте создадим структуру, поддерживающую этот протокол. Для этого нужно указать его при объявлении структуры после двоеточия. Таким образом можно перечислить через запятую несколько протоколов.

```
// --- Протоколы CODE SNIPPET #2 --- //

struct MyStruct: MyProtocol {

}
```

Объявим более полезный протокол. Добавим в него несколько требований. Для этого нужно их указать так же как при объявлении в структуре или классе. После типа в фигурных скобка указывается может ли это свойство быть только для чтения или нужно дать доступ и на установку значения.

```
// --- Протоколы CODE SNIPPET #3 --- //

protocol MyProtocolWithProperties {
        var settableProperty: Int { set get }
        var gettableProperty: Double { get }
}
```

Добавим новую структуру. В ней объявим пару свойств для поддержки протокола.

```
// --- Протоколы CODE SNIPPET #4 --- //
```

```
struct MyStructWithProperties: MyProtocolWithProperties {
	var settableProperty: Int {
		set {
		}

		get {
			return 5
		}
	}

	var gettableProperty: Double
}
```

Протокол не может требовать вычисляемое или обычное свойство. Поэтому мы в структуре можем использовать и то и другое. Обратите также внимание на то, что можно определить сеттер для свойства даже если протоколе указано read only.

В протокол можно добавить требования не только для экземпляров, но и для самих типов. Для этого нужно добавить ключевое слово static перед свойством.

```
// --- Протоколы CODE SNIPPET #5 --- //

protocol MyProtocolWithTypeProperty {
   static var typeProperty: Int { get }
}
```

Давайте объявим протокол с требованиями к методам.

```
// --- Протоколы CODE SNIPPET #6 --- //

protocol MyProtocolWithMethods {
	mutating func instanceMethod(parameter: Int)
	static func typeMethod() -> Double
}
```

Они похожи на объявление методов в структурах и классах, но без фигурных скобок и реализации. Также можно использовать слово static для методов типов, но значения по умолчанию указывать нельзя.

Добавим структуру с поддержкой нескольких протоколов.

```
// --- Протоколы CODE SNIPPET #7 --- //
struct MyStructWithPropertiesAndMethods: MyProtocolWithProperties,
MyProtocolWithMethods {
	var settableProperty: Int
```

```
        var gettableProperty: Double

        mutating func instanceMethod(parameter: Int) {
        gettableProperty = Double(parameter)
        }

        static func typeMethod() -> Double {
        return 4.0
        }
}
```

Реализация методов может быть любой. Протокол требует только наличия самого метода, но не логики в нем. Т.к. мы добавили mutating при объявлении протокола мы можем изменять значения свойств в этом методе.

Помимо обычных методов протокол может требовать наличие инициализатора.

```
// --- Протоколы CODE SNIPPET #8 --- //

protocol MyProtocolWithInit {
        init(paramenter: Int)
}
```

Объявляется он также - без фигурных скобок и реализации.

Однако есть одна особенность. В классе поддерживающем этот протокол нужно обязательно объявлить такой инициализатор как required.

```
// --- Протоколы CODE SNIPPET #9 --- //

class MyClassWithInit: MyProtocolWithInit {
        let someConst: Int

        required init(paramenter: Int) {
        someConst = paramenter
        }
}
```

Это нужно для того, чтобы все наследники тоже предоставили собственную реализацию. Подробнее о наследовании и инициализаторах смотрите в соответствующих лекциях.

Несмотря на то, что протоколы не включают в себя никакую имплементацию, а только требования мы все равно можем использовать их как любой другой тип в Swift: можно объявить переменную или константу этого типа, можно принимать и возвращать их из функций и т.д.

Объявим переменную типа MyProtocolWithProperties.

// --- Протоколы CODE SNIPPET #10 --- //

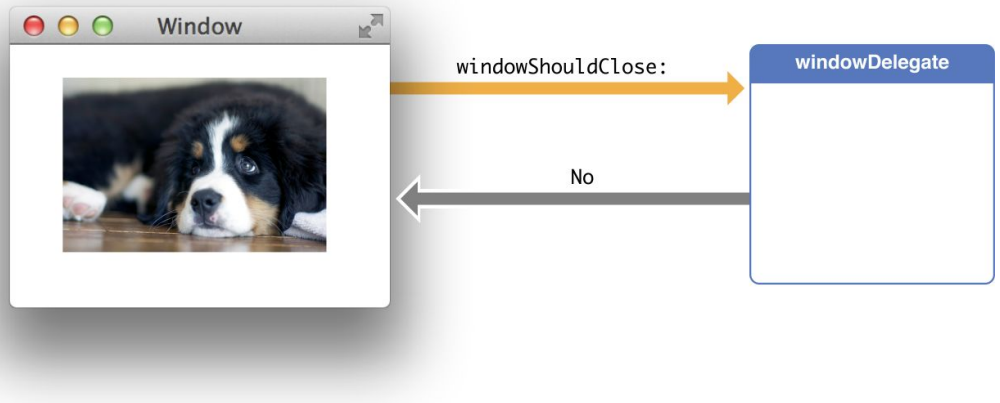
```
var someObject: MyProtocolWithProperties = MyStructWithProperties(gettableProperty: 5.0)
someObject.settableProperty


someObject = MyStructWithPropertiesAndMethods(settableProperty: 1,
                                              gettableProperty: 2.0)
someObject.settableProperty
```

Присвоим ей экземпляр структуры MyStructWithProperties. При этом информация о том, экземпляр какого класса или структуры лежит в этой переменной теряется. Вызывающая сторона может обратиться только к тем свойствам и методам, которые есть в требованиях протокола.

Мы можем присвоить этой переменной экземпляр другой структуры. При этом для вызывающей стороны ничего не изменится. Известно лишь то, что у этого объекта будут свойства gettableProperty и settableProperty.

Это один из вариантов использования протоколов. Он часто встречается в разработке под iOS в виде архитектурного паттерна - [делегат](). Реализуем пример делегата.



Для начала опишем класс, который будет обрабатывать массив строк.

// --- Протоколы CODE SNIPPET #11 --- //

```
class StringProcessor {
        var delegate: StringProcessorDelegate?
```

```
        func process(_ strings: [String]) -> String {
        guard strings.count > 0, let delegate = delegate else {
        return ""
        }

        var temp = [String]()
        for string in strings {
        let t = delegate.transform(string)
        temp.append(t)
        }

        return temp.joined(separator: delegate.concatenateSeparator)
        }
}
```

У него будет свойство хранящее делегат. В методе process он каждую строку трансформирует с помощью делегата и складывает результат в одну строку используя разделитель.

В требованиях в протоколе укажем метод для трансформации и разделитель в виде свойства.

```
// --- Протоколы CODE SNIPPET #12 --- //

protocol StringProcessorDelegate {
        var concatenateSeparator: String { get }

        func transform(_ string: String) -> String
}
```

Объявим также несколько классов поддерживающих этот протокол.

```
// --- Протоколы CODE SNIPPET #13 --- //

class UpperCaseStringProcessorDelegate: StringProcessorDelegate {
        var concatenateSeparator: String {
        return " "
        }

        func transform(_ string: String) -> String {
        return string.uppercased()
        }
}


class FirstLetterStringProcessorDelegate: StringProcessorDelegate {
```

```
        var concatenateSeparator: String {
        return ""
        }

        func transform(_ string: String) -> String {
        guard let firstCharacter = string.first else {
        return ""
        }
        return String(firstCharacter)
        }
}
```

Первый будет трансформировать все строки в верхнем регистре. Разделителем будет пробел.

Второй возвращает из трансформации только первый символ в строке. Разделитель у него пустая строка.

Попробуем все в действии. Создадим процессор, его делегаты и тестовый массив.

```
// --- Протоколы CODE SNIPPET #14 --- //

var processor = StringProcessor()
let upperCaseDelegate = UpperCaseStringProcessorDelegate()
let firstLetterDelegate = FirstLetterStringProcessorDelegate()
let testArray = ["This", "is", "simple", "test", "strings"]
```

Сначала просто вызовем обработку без делегатов. Получим ожидаемую пустую строку.

```
// --- Протоколы CODE SNIPPET #15 --- //

processor.process(testArray) // ""
```

Выставим первый делегат в процессоре. В результате получим строку состоящую из заглавных букв.

```
// --- Протоколы CODE SNIPPET #16 --- //

processor.delegate = upperCaseDelegate
processor.process(testArray) // "THIS IS SIMPLE TEST STRINGS"
```

Попробуем тоже самое со вторым. Получим строку состоящую из первых букв тестовых слов.

```
// --- Протоколы CODE SNIPPET #17 --- //
```

```
processor.delegate = firstLetterDelegate
processor.process(testArray) // "Tists"
```

Как видите мы вынесли логику из процессора в делегат. Это дает нам возможность легко изменять поведение, просто поменяв один делегат на другой. Со стороны процессора при этом ничего не меняется. Он знает только, что его делегат будет предоставлять ему методы, перечисленные в протоколе, с ними он и взаимодействует.

Помимо простой переменной мы можем объявить и целую коллекцию объектов, поддерживающих какой-то протокол. Создадим массив делегатов процессора.

```
// --- Протоколы CODE SNIPPET #18 --- //

var arrayOfDelegates = [StringProcessorDelegate]()
```

Теперь мы можем положить в него любой объект поддерживающий этот протокол.

```
// --- Протоколы CODE SNIPPET #19 --- //

arrayOfDelegates.append(upperCaseDelegate)
arrayOfDelegates.append(firstLetterDelegate)

for delegate in arrayOfDelegates {
        delegate.concatenateSeparator
}
```

Тип объектов будет соответствовать протоколу. Вы не будете знать, что за объект вы получили из массива. Конечно, в Swift есть приведение типов, но если у вас при написании кода возникает такая необходимость значит вы что-то делаете не так.

Apple называет Swift протоколо-ориентированным языком программирования. Протоколы это не просто абстрактный интерфейс класса или структуры. Добавляя поддержку протоколов в какой-либо тип мы добавляем ему определенную функциональность. Например, Equatable для проверки на равенство, ExpressibleByIntegerLiteral для инициализации типа с помощью целочисленного литерала, Hashable для получения хэш-суммы объекта и т.д. Объявляя какой-нибудь метод мы просто перечисляем поддержку какой функциональности мы ждем от получаемого объекта. При этом нам уже не важен его тип. В лекции по расширению функциональности мы покажем как добавить поддержку протокола уже существующему типу. А пока посмотрим как использовать композицию протоколов.

Композиция протоколов - это объединение нескольких протоколов в один тип. Их еще называют экзистенциальными типами. Объявляются они перечислением протоколов через амперсанд.

// --- Протоколы CODE SNIPPET #20 --- //

```
let compoundObject: MyProtocolWithProperties & MyProtocolWithMethods
compoundObject = MyStructWithPropertiesAndMethods(settableProperty: 1,
gettableProperty: 2.0)
```

Переменная compoundObject может содержать только объект поддерживающий оба протокола.

В Swift 4 добавили возможность объявить экзистенциальный тип, содержащий не только протоколы, но и класс. Это может очень пригодиться при работе с системными классами. В следующем примере мы объявляем переменную с таким типом.

// --- Протоколы CODE SNIPPET #21 --- //

```
let compoundClassObject: UIView & MyProtocol
```

Она может содержать только экземпляры класса UIView либо ее сабклассов. При этом они обязательно должны поддерживать протокол MyProtocol. Вы еще встретите подобные конструкции в следующих лекциях и будет понятнее как это использовать.

Мы можем проверить поддерживает ли объект какой-то протокол или привести, например, AnyObject к нужному протоколу. Делается это так же как с любым другим типом - с помощью операторов is, as!, as?.

// --- Протоколы CODE SNIPPET #22 --- //

```
let arrayOfAny: [Any] = [4, firstLetterDelegate, "String"]

for object in arrayOfAny {
        if let processorDelegate = object as? StringProcessorDelegate {
        processor.delegate = processorDelegate
        processor.process(testArray) // "Tists"
        }
}
```

Рассмотрим простой пример. У нас есть массив объектов, который по какой-то причине имеет тип [Any]. Но мы знаем, что в нем должны быть объекты, которые поддерживающие какой-то протокол.

Воспользуемся as? и if let для безопасного преобразования типа. Даже если в этот массив попало что-то другое, как в нашем примере, то ничего страшного не произойдет. Не стоит использовать as! даже если уверены, что ничего лишнего вам прийти не могло.

В Swfit все требования протокола обязательны к исполнению. Если вы хотите иметь возможность использовать опциональные протоколы, как в Objective-C, то вам придется воспользоваться атрибутом @objc. Он был добавлен в Swift для обеспечения взаимодействия с Objective-C.

Объявим протокол, добавив к нему @objc. Теперь он будет доступен для Objective-C кода. Все опциональные требования тоже должны быть помечены этим атрибутом.

```
// --- Протоколы CODE SNIPPET #23 --- //

@objc protocol ObjCProtocol {
	@objc optional func optionalFunc() -> Int
	func normalSwiftRequiredFunc()
}
```

Методы и свойства, объявленные таким образом, будут иметь опциональный тип. Т.е. в нашем примере это будет (() -> Int)? . Обратите внимание, что опциональным стал не возвращаемый тип, а сам метод.

У опциональных протоколов есть ограничения. Они могут поддерживаться только классами из Objective-C, либо Свифтовыми классами помеченными атрибутом @objc. При этом структуры и перечисления не могут поддержать такой протокол.

Объявим такой класс.

```
// --- Протоколы CODE SNIPPET #24 --- //

@objc class MyObjcExposedClass: NSObject, ObjCProtocol {
	func normalSwiftRequiredFunc() {
	}
}

let objcClass: ObjCProtocol = MyObjcExposedClass()
let res = objcClass.optionalFunc?()
res // nil
```

Наследование мы рассмотрим в будущих лекциях, но сейчас мы обязаны воспользоваться им. Синтаксис наследования в Swift похож на другие языки программирования и скорее всего покажется вам знакомым. Тут мы объявляем класс наследник NSObject и добавляем ему поддержку нашего опционального протокола.

NSObject это основной базовый класс в Objective-C. Он него наследуется большинство других классов. Скорее всего вы будете работать либо с ним, либо с одним из его сабклассов.

Как видите мы не добавили опциональный метод в наш класс. При вызове нужно учитывать такую ситуацию. Для этого укажем после имени метода ?. В этом случае ничего вызываться не будет и в результате будет nil. Обратите внимание, что тип переменной указан явно и является ObjCProtocol. Если бы переменная была MyObjcExposedClass, то Swift мог бы определить, что нужный метод отсутствует и не дал бы выполнить код.