Практическая работа № 14

*Протоколы и расширения*

Цель работы:

1. Изучить и научиться работать с *протоколами и расширениями*.
2. Изучить их особенности применения.
3. Научиться использовать их в программе.

Краткие теоретические сведения

*Протоколы*

Протокол определяет образец методов, свойств или другие требования, которые соответствуют определенному конкретному заданию или какой-то функциональности. Протокол фактически не предоставляет реализацию для любого из этих требований, он только описывает как реализация должна выглядеть. Протокол может быть принят классом, структурой или перечислением для обеспечения фактической реализации этих требований. Любой тип, который удовлетворяет требованиям протокола, имеет указание соответствовать этому протоколу или другими словами реализовать данный протокол.

*Для тех кто пришел из мира C++/C#/Java – это по другому названные интерфейсы.*

**Синтаксис протокола**

Определение протокола очень похоже на то, как вы определяете классы, структуры и перечисления:

protocol SomeProtocol {

// определение протокола…

}

Пользовательские типы утверждают, что они принимают протокол, когда они помещают имя протокола после имени типа и разделяются с этим именем двоеточием, то есть указывают эти протоколы как часть их определения. После двоеточия вы можете указывать множество протоколов, перечисляя их имена через запятую:

struct SomeStructure: FirstProtocol, AnotherProtocol {

// определение структуры…

}

Если у класса есть суперкласс, то вписывайте имя суперкласса до списка протоколов, которые он принимает, также разделите имя суперкласса и имя протокола запятой:

class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol {

// определение класса…

}

}

Перед требуемыми свойствами типов пишете префикс static, когда вы определяете их в протоколе. Это правило распространяется даже тогда, когда требование свойств может иметь как префикс static так и префикс class, когда мы реализуем их в классах:

protocol AnotherProtocol {

static var someTypeProperty: Int { get set }

}

Пример протокола с единственным требуемым свойством экземпляра:

protocol FullyNamed {

var fullName: String { get }

}

Протокол FullyNamed требует у соответствующего ему типа предоставить полное имя. Протокол больше не уточняет ничего, кроме того, что тип этого свойства должен быть в состоянии предоставить свое полное имя. Протокол утверждает, что любой тип FullyNamed должен иметь свойство fullName, значение которого может быть получено, и это значение должно быть типа String.

Ниже приведен пример структуры, которая принимает и полностью соответствует протоколу FullyNamed:

struct Person: FullyNamed {

var fullName: String

}

let john = Person(fullName: "John Appleseed")

// john.fullName равен "John Appleseed"

Следующий пример определяет протокол с единственным требуемым методом экземпляра:

protocol RandomNumberGenerator {

func random() -> Double

}

Этот протокол RandomNumberGenerator требует любой соответствующий ему тип иметь метод экземпляра random, который при вызове возвращает значение типа Double. Хотя это и не указано как часть протокола, но предполагается, что значение будет числом от 0.0 и до 1.0 (не включительно).

Протокол RandomNumberGenerator не делает никаких предположений по поводу того, как будет находиться это случайное число, он просто требует генератор предоставить стандартный способ генерации нового рандомного числа.

**Основные особенности протоколов в Swift:**

* Протоколы обеспечивают множественное наследование
* Протоколы не могут хранить состояние
* Протоколы могут быть унаследованы другими протоколами
* Протоколы могут применяться к структурам (struct), классам(class) и перечислениям (enum), определяя функционал типов
* Дженерик протоколы (Generic-protocol) позволяют задавать сложные зависимости между типами и протоколами во время их наследования
* Протоколы не определяют «сильные» и «слабые» ссылки на переменные
* В расширениях к протоколам можно описывать конкретные реализации методов, и вычисляемых переменных (computed values)
* Классовые протоколы разрешают себя наследовать только классам

## *Реализации класса требований инициализатора протокола*

*Назначенный или удобный инициализатор позволяет пользователю инициализировать протокол для соответствия его стандарту с помощью зарезервированного ключевого слова required.*

*class SomeClass: SomeProtocol {*

*required init(someParameter: Int) {*

*// initializer implementation statements*

*}*

*}*

*protocol tcpprotocol {*

*init(aprot: Int)*

*}*

*class tcpClass: tcpprotocol {*

*required init(aprot: Int) {*

*}*

*}*

*Соответствие протокола обеспечивается во всех подклассах для явной или унаследованной реализации с помощью модификатора required.*

*Когда подкласс переопределяет требование инициализации суперкласса, он задается ключевым словом модификатора override.*

*protocol tcpprotocol {*

*init(no1: Int)*

*}*

*class mainClass {*

*var no1: Int // local storage*

*init(no1: Int) {*

*self.no1 = no1 // initialization*

*}*

*}*

*class subClass: mainClass, tcpprotocol {*

*var no2: Int*

*init(no1: Int, no2 : Int) {*

*self.no2 = no2*

*super.init(no1:no1)*

*}*

*// Requires only one parameter for convenient method*

*required override convenience init(no1: Int) {*

*self.init(no1:no1, no2:0)*

*}*

*}*

*let res = mainClass(no1: 20)*

*let print = subClass(no1: 30, no2: 50)*

*print("res is: \(res.no1)")*

*print("res is: \(print.no1)")*

*print("res is: \(print.no2)")*

*Когда мы запускаем вышеуказанную программу, используя площадку, мы получаем следующий результат —*

*res is: 20*

*res is: 30*

*res is: 50*

***Расширения***

Расширения добавляют новую функциональность существующему типу класса, структуры или перечисления. Это включает в себя возможность расширять типы, к исходным кодам которых у вас нет доступа (известно как ретроактивное моделирование).

Расширения в Swift могут:

* Добавлять вычисляемые свойства и вычисляемые свойства типа
* Определять методы экземпляра и методы типа
* Предоставлять новые инициализаторы
* Определять сабскрипты (индексы)
* Определять новые вложенные типы
* Обеспечить соответствие существующего типа протоколу

Расширения могут добавлять новую функциональность типу, но они не могут переписать существующую функциональность.

**Синтаксис расширений**

Расширение объявляется с помощью ключевого слова extension:

extension SomeType {

// описываем новую функциональность для типа SomeType

}

Расширение может расширить существующий тип для того, чтобы он соответствовал одному или более протоколам. Там где это имеет место, имена протоколов записываются точно так же, как и в случае с классами или структурами:

extension SomeType: SomeProtocol, AnotherProtocol {

// реализация требования протокола тут

}

В примере мы добавляем пять вычисляемых свойств экземпляра во встроенный тип Double языка Swift, для обеспечения работы данного типа с единицами длины:

extension Double {

var km: Double { return self \* 1\_000.0 }

var m: Double { return self }

var cm: Double { return self / 100.0 }

var mm: Double { return self / 1\_000.0 }

var ft: Double { return self / 3.28084 }

}

let oneInch = 25.4.mm

print("Один дюйм - это \(oneInch) метра")

// Выведет "Один дюйм- это 0.0254 метра"

let threeFeet = 3.ft

print("Три фута - это \(threeFeet) метра")

// Выведет "Три фута - это 0.914399970739201 метра"

Эти вычисляемые свойства объясняют, что тип Double должен считаться как конкретная единица измерения длины. Хотя они реализованы как вычисляемые свойства, имена этих свойств могут быть добавлены к литералу чисел с плавающей точкой через точечный синтаксис, как способ использовать значения литерала для проведения преобразований длины.

В этом примере, значение 1.0 типа Double отображает “один метр”. Это причина, по которой m возвращает self, что равно 1.m, то есть посчитать Double от числа 1.0.

Расширение работает в значительной степени подобно абстрактным классам при рассмотрении функциональности, которую вы хотите получить во всех классах, реализующих некоторый протокол (без наследования от базового общего класса).

* protocol FooProtocol {
* func doSomething()
* }
* extension FooProtocol {
* func doSomething() {
* print("Hi")
* }
* }
* class Foo: FooProtocol {
* func myMethod() {
* doSomething() // By just implementing the protocol this method is available
* }
* }

Задание для самостоятельного выполнения

1. Создайте протокол для класса в соответствие с вариантом задания, реализуйте свойства типа (любые комбинации **get/set**), реализуйте в нем инициализатор. Выведите экземпляр класса в консоль.
2. Создайте расширение, выводящее наименование численного значения Вашего класса.
3. Вывести экземпляр объекта в консоль.

Варианты

1. Автобус (номер маршрута, название начальной и конечной остановки, количество остановок по маршруту)
2. Книга (название книги, автор, издательство, количество страниц)
3. Файл (имя, время создания, дата создания, размер)
4. Спортсмен (ФИО, вид спорта, сколько лет занимается)
5. Студент (ФИО, группа, средний балл зачетки)
6. Фирма (название, вид услуг, годовой оборот)
7. Компьютер (производитель, частота процессора, количество ядер, объем памяти)
8. Спорт (вид спорта, количество занимающихся в городе, средняя стоимость комплекта инвентаря)
9. Автомобиль (марка, фирма-производитель, год выпуска, пробег)
10. Статья (название, автор, название журнала, год и месяц издания)