**Лабораторная работа №8**

**РАСШИРЕНИЯ, ПРОТОКОЛЫ**

**Расширения**

Расширения позволяют добавить новую функциональность к существующему объектному типу (классу, структуре или перечислению), а также к протоколу. Более того, вы можете расширять типы данных, доступа к исходным кодам которых у вас нет (например, типы, предоставляемые фреймворками, или фундаментальные для Swift типы данных).

Перечислим возможности расширений:

* добавление вычисляемых свойств экземпляра и вычисляемых свойств типа (static);
* определение методов экземпляра и методов типа;
* определение новых инициализаторов, сабскриптов и вложенных типов;
* обеспечение соответствия существующего типа протоколу.

Расширения могут добавлять новый функционал к типу, но не могут изменять уже существующий. Суть расширения состоит исключительно в наращивании возможностей, но не в их изменении.

СИНТАКСИС

extension ИмяРасширяемогоТипа {

// описание новой функциональности для типа SomeType

}

Новая функциональность, добавляемая расширением, становится доступной всем экземплярам расширяемого объектного типа вне зависимости от того, где эти экземпляры объявлены.

Вычисляемые свойства в расширениях

Расширения могут добавлять вычисляемые свойства экземпляра и вычисляемые свойства типа в определенный тип данных. Рассмотрим пример расширения функционала типа Double, создав в нем ряд новых вычисляемых свойств и обеспечив тип Double возможностью работы с единицами длины.

extension Double {

var km: Double { return self \* 1000.0 }

var m: Double { return self }

var cm: Double { return self / 100.0 }

var mm: Double { return self / 1000.0 }

var ft: Double { return self / 3.28084 }

}

let oneInch = 25.4.mm

print("Один фут — это \(oneInch) метра")

// выводит "Один фут — это 0.0254 метра"

let threeFeet = 3.ft

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

// выводит "Три фута — это 0.914399970739201 метра"

Созданные вычисляемые свойства позволяют использовать дробные числа как конкретные единицы измерения длины. Добавленные новые свойства могут применяться для параметров и литералов типа Double. В данном примере подразумевается, что значение 1 .0 типа Double отражает величину один метр. Именно поэтому свойство m возвращает значение self. Другие свойства требуют некоторых преобразований перед возвращением значений. Один километр — это то же самое, что 1000 метров, поэтому при запросе свойства km возвращается результат выражения self \* 1000. Чтобы после определения новых вычисляемых свойств использовать всю их мощь, требуется создавать и геттеры, и сеттеры.

ПРИМЕЧАНИЕ. Расширения могут добавлять только новые вычисляемые свойства. При попытке добавить хранимые свойства или наблюдателей свойств происходит ошибка.

Инициализаторы в расширениях

Расширения могут добавлять инициализаторы к уже существующему типу. Таким образом, вы можете расширить существующие типы, например, для обработки экземпляров ваших собственных типов в качестве входных аргументов.

ПРИМЕЧАНИЕ . Для классов расширения могут добавлять только новые *вспомогательные инициализаторы*. Попытка добавить назначенный инициализатор или деинициализатор ведет к ошибке. В качестве примера напишем инициализатор для типа Double. В этом примере создается структура, описывающая линию на плоскости. Необходимо реализовать инициализатор, принимающий в качестве входного аргумента экземпляр линии и устанавливающий значение, соответствующее длине линии.

import UIKit

// сущность "линия"

struct Line{

var pointOne: (Double, Double)

var pointTwo: (Double, Double)

}

// расширения для Double

extension Double {

init(line: Line){

self = sqrt(pow((line.pointTwo.0 - line.pointOne.0),2) + pow((line.pointTwo.1 - line.pointOne.1),2))

}

}

var myLine = Line(pointOne: (10,10), pointTwo: (14,10))

var lineLength = Double(line: myLine) // выводит 4

Библиотека UIKit обеспечивает доступ к математическим функциям sqrt(\_:) и pow(\_:\_:) (соответственно квадратный корень и возведение в степень), которые требуются для вычисления длины линии на плоскости . Структура Line описывает сущность «линия», в свойствах которой указываются координаты точек ее начала и конца. Созданный в расширении инициализатор принимает на входе экземпляр класса Line и на основе значений его свойств вычисляет требуемое значение. При разработке нового инициализатора в расширении будьте крайне внимательны к тому, чтобы к завершению инициализации каждое из свойств имело определенное значение.

Методы в расширениях

Следующей рассматриваемой функцией расширений является создание новых методов в расширяемом типе данных. Рассмотрим пример. В этом примере путем расширения типа Int мы добавляем метод repetitions, принимающий на входе замыкание типа () -> (). Данный метод предназначен для того, чтобы выполнять переданное замыкание столько раз, сколько указывает собственное значение целого числа.

extension Int {

func repetitions(task: () -> ()) {

for \_ in 0..<self {

task()

}

}

}

3.repetitions(task: {

print("Swift")

})

/\* выводит

Swift

Swift

Swift \*/

Для изменения свойств перечислений и структур реализуемыми расширением методами необходимо не забывать использовать модификатор mutating. В следующем примере реализуется метод square(), который возводит в квадрат собственное значение экземпляра. Так как тип Int является структурой, то для изменения собственного значения экземпляра необходимо использовать ключевое слово mutating.

extension Int {

mutating func square() {

self = self \* self

}

}

var someInt = 3

someInt.square()

Сабскрипты в расширениях

Помимо свойств, методов и инициализаторов, расширения позволяют создавать новые сабскрипты. Создаваемое расширение типа Int реализует новый сабскрипт, который позволяет получить определенную цифру собственного значения экземпляра. В сабскрипте указывается номер позиции числа, которое необходимо вернуть.

extension Int {

subscript( digitIndex: Int) -> Int {

var decimalBase = 1

while digitIndex > 0 {

decimalBase \*= 10 - digitIndex

}

return (self / decimalBase) % 10

}

}

746381295[0]

// возвращает 5

746381295[1]

// возвращает 9

Если у числа отсутствует цифра с запрошенным индексом, возвращается 0, что не нарушает логику работы.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Задание 8.1**

1. *Сделайте расширение для типа Int с методами проверки четности, нечетности, реализации нового сабскрипта – который возвращает количество цифр в числе в соответсвии с заданной цифрой в сабскрипте (например, 222[1] 🡪 0 , 222[2]🡪 3).*
2. *Добавьте через расширение новый инициализатор в классе Bug, а также метод, возвращающий количество дней устранения бага (если баг не закрыт, т.е. любой стаутс отличный от Closed, считается как текущая дата – дата открытия бага), метод для переоткрытия бага (меняется статус на Reopened и обновляется дата).*

**Протоколы**

Протоколы содержат перечень свойств, методов и сабскриптов, которые должны быть реализованы в объектном типе. Другими словами, они содержат требования к наличию определенных элементов внутри типа данных. Протокол сам непосредственно не реализует какой-либо функционал, он лишь является своеобразным набором правил и требований к типу. Любой объектный тип данных может принимать протокол. Наиболее важной функцией протокола является обеспечение целостности объектных типов путем указания требований к их реализации. Протоколы объявляются независимо от каких-либо элементов программы, так же как и объектные типы данных.

СИНТАКСИС

protocol ИмяПротокола {

// тело протокола

}

Для объявления протокола используется ключевое слово protocol, после которого указывается имя создаваемого протокола. Для того чтобы принять протокол к исполнению каким-либо объектным типом, необходимо написать его имя через двоеточие сразу после имени реализуемого типа:

struct ИмяПринимающейСтруктуры: ИмяПротокола{

// тело структуры

}

После указания имени протокола при объявлении объектного типа данный тип обязан выполнить все требования протокола. Вы можете указать произвольное количество принимаемых протоколов.

Если класс не только принимает протоколы, но инаследует некоторый класс, то имя супер класса необходимо указать первым, а за ним через запятую — список протоколов:

class ИмяПринимающегоКласса: ИмяСуперКласса,

Протокол1, Протокол2{

// тело класса

}

Требуемые свойства

Протокол может потребовать соответствующий ему тип реализовать свойство экземпляра или свойство типа (static), имеющее конкретные имя и тип данных. При этом протокол не указывает на вид свойства (хранимое или вычисляемое). Также могут быть указаны требования к *доступности и изменяемости параметра*. Если у свойства присутствует требование доступности и изменяемости, то в качестве данного свойства не могут выступать константа или вычисляемое свойство «только для чтения». Требование доступности обозначается с помощью конструкции {get}, а требование доступности и изменяемости — с помощью конструкции {get set}.

protocol SomeProtocol {

var mustBeSettable: Int { get set }

var doesNotNeedToBeSettable: Int { get }

}

Протокол SomeProtocol требует, чтобы в принимающем типе были реализованы два изменяемых (var) свойства типа Int. При этом свойство mustBeSettable должно быть и доступным для чтения, и изменяемым, а свойство doesNotNeedToBeSettable — как минимум доступным для чтения. Протокол *определяет минимальные требования* к типу, то есть тип данных обязан реализовать все, что описано в протоколе, но он может не ограничиваться этим набором элементов. Так, для свойства doesNotNeedToBeSettable может быть реализован не только геттер, но и сеттер. Для указания в протоколе на свойство типа необходимо использовать модификатор static перед ключевым словом var. Данное требование выполняется даже в том случае, если протокол принимается структурой или перечислением.

protocol AnotherProtocol {

static var someTypeProperty: Int { get set }

}

В данном примере свойство типа someTypeProperty должно быть обязательно реализовано в принимающем типе данных. В следующем примере мы создадим протокол и принимающий его требования класс.

protocol FullyNamed {

var fullName: String { get }

}

struct Person: FullyNamed {

var fullName: String

}

let john = Person(fullName: "John “)

В данном примере определяется протокол FullyNamed, который обязывает структуру Person иметь доступное свойство fullName типа String .

Требуемые методы

Протокол может требовать реализации определенного метода экземпляра или метода типа. Форма записи для этого подобна указанию требований к реализации свойств. Для требования реализации метода типа необходимо использовать модификатор static. Также протокол может описывать изменяющий метод. Для этого служит модификатор mutating.

ПРИМЕЧАНИЕ. Если вы указали ключевое слово mutating перед требованием метода, то вам уже не нужно указывать его при реализации метода в классе. Данное ключевое слово требуется только в реализации структур.

При реализации метода в типе данных необходимо в точности соблюдать все требования протокола, в частности имя метода, наличие или отсутствие входных аргументов, тип возвращаемого значения, модификаторы

protocol RandomNumberGenerator {

func random() -> Double

static func getRandomString()

mutating func changeValue(newValue: Double)

}

Требуемые инициализаторы

Дополнительно протокол может предъявлять требования к реализации инициализаторов. Необходимо писать инициализаторы точно так же, как вы пишете их в объектном типе, опуская фигурные скобки и тело инициализатора. Требования к инициализаторам могут быть выполнены в соответствующем классе в форме назначенного или вспомогательного инициализатора. В любом случае перед объявлением инициализатора необходимо указывать модификатор required. Это гарантирует, что вы реализуете указанный инициализатор во всех подклассах данного класса.

protocol Named{

init(name: String)

}

class Person : Named {

var name: String

required init(name: String){

self.name = name

}

}

Протокол в качестве типа данных

Протокол сам по себе не несет какой-либо функциональной нагрузки, он лишь содержит требования к реализации объектных типов. Тем не менее протокол является полноправным типом данных. Используя протокол в качестве типа данных, вы указываете на то, что записываемое в данное хранилище значение должно иметь тип данных, который соответствует указанному протоколу. Так как протокол является типом данных, вы можете организовать проверку соответствия протоколу с помощью оператора is. При проверке соответствия возвращается значение true, если проверяемый экземпляр соответствует протоколу, и false в противном случае.

Расширение и протоколы

Расширения могут взаимодействовать не только с объектными типами, но и с протоколами. Вы можете использовать расширения для добавления требований по соответствию некоторого объектного типа протоколу. Для этого в расширении после имени типа данных через двоеточие необходимо указать список новых протоколов

protocol TextRepresentable {

func asText() -> String

}

extension Int: TextRepresentable {

func asText() -> String {

return String(self)

}

}

5.asText()

В данном примере протокол TextRepresentable требует, чтобы в принимающем объектном типе был реализован метод asText(). С помощью расширения мы добавляем требование о соответствии типа Int к данному протоколу, при этом, поскольку где-то ранее был реализован сам тип данных, в обязательном порядке указывается реализация данного метода.

С помощью расширений мы можем не только указывать на необходимость соответствия новым протоколам, но и расширять сами протоколы, поскольку протоколы являются полноценными типами данных. При объявлении расширения необходимо использовать имя протокола, а в его теле указывать набор требований с их реализациями. После расширения протокола описанные в нем реализации становятся доступны в экземплярах всех классов, которые приняли данный протокол к исполнению. Напишем расширение для реализованного ранее протокола TextRepresen-table

extension TextRepresentable {

func description() -> String {

return "Данный тип поддерживает протокол TextRepresentable"

}

}

5.description()// Данный тип поддерживает протокол TextRepresentable

Расширение добавляет новый метод в протокол TextRepresentable. При этом ранее мы указали, что тип Int соответствует данному протоколу. В связи с этим появляется возможность обратиться к указанному методу для любого значения типа Int.

Наследование протоколов

Протокол может наследовать один или более других протоколов. При этом он может добавлять новые требования поверх наследуемых, — тогда тип, принявший протокол к реализации, будет вынужден выполнить требования всех протоколов в структуре. При наследовании протоколов используется тот же синтаксис, что и при наследовании классов.

Классовые протоколы

Вы можете ограничить протокол таким образом, чтобы его могли принимать к исполнению исключительно классы (а не структуры и перечисления). Для этого у протокола в списке наследуемых протоколов необходимо указать ключевое слово class. Данное слово всегда должно указываться на первом месте в списке наследования.

protocol SubProtocol: class, SuperProtocol {

func someMethod()

}

Композиция протоколов

Иногда бывает удобно требовать, чтобы тип соответствовал не одному, а нескольким протоколам. В этом случае, конечно же, можно создать новый протокол, наследовать в него несколько необходимых протоколов и задействовать имя только что созданного протокола. Однако для решения данной задачи лучше воспользоваться композицией протоколов, то есть скомбинировать несколько протоколов.

СИНТАКСИС

Протокол1 & Протокол2 ...

Для композиции протоколов необходимо указать имена входящих в данную композицию протоколов, разделив их оператором & (амперсанд).

protocol Named {

var name: String { get }

}

protocol Aged {

var age: Int { get }

}

struct Person: Named, Aged {

var name: String

var age: Int

}

func wishHappyBirthday(celebrator: Named & Aged) {

print("С Днем рождения, \(celebrator.name)! Тебе уже\(celebrator.age)!")

}

let birthdayPerson = Person(name: "Harry", age: 11)

wishHappyBirthday(celebrator: birthdayPerson)

//С Днем рождения, Harry! Тебе уже11!

В данном примере объявляются два протокола: Named и Aged. Созданная структура принимает оба протокола и в полной мере выполняет их требования. Входным аргументом функции wishHappyBirthday(celebrator:) должно быть значение, которое удовлетворяет обоим протоколам. Таким значением является экземпляр структуры Person, который мы и передаем.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Задание 8.2**

1. *Создайте протокол BugTracker с коллекцией Bug и следующими функциями – создать, назначить, изменить статус Bug ( см. л.р. №5), сортировка багов.*
2. *Реализуйте требуемые функции BugTracker в классе JIRA.*
3. *Расширьте протокол BugTracker функцией подготовки отчета.*
4. *Проверье работу JIRA создав его инстанс.*

**Вопросы**

1. Для каких типов можно использовать расширение?
2. Что можно определять в расширениях?
3. В чем состоит назначение протокола?
4. Приведите пример протокола с требуемым свойством. Как задать требование доступности и изменяемости?
5. Приведите пример расширения протокола.
6. Что такое классовые протоколы?
7. Как определить и где можно использовать композицию протоколов?
8. Объясните зачем и как в протоколе использовать связанный тип? Как использовать такой протокол?