**Extension (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.

Расширения в Swift могут:

1. Добавлять вычисляемые свойства и вычисляемые свойства типа

2. Определять методы экземпляра и методы типа

3. Предоставлять новые инициализаторы

4. Определять сабскрипты (индексы)

5. Определять новые вложенные типы

6. 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:

class A {}

extension A {

}

extension A: Protocol {

}

Если вы определяете расширение для добавления новой функиональности существующему типу, то новая функциональность будет доступна всем экземплярам этого типа, даже если они были созданы до того, как было определено расширение.

Вычисляемые свойства в расширениях

Расширения могут добавлять вычисляемые свойства экземпляра и вычисляемые свойства типа к существующим типам. 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

}

}

}

Расширения могут добавлять новые вычисляемые свойства, но они не могут добавить свойства хранения или наблюдателей свойства к уже существующим свойствам.

Инициализаторы в расширениях

Расширения могут добавить новые инициализаторы существующему типу. Это позволяет вам расширить другие типы для принятия ваших собственных типов в качестве параметров инициализатора, или для обеспечения дополнительных опций инициализации, которые не были включены как часть первоначальной реализации типа.

Расширения могут добавлять вспомогательные инициализаторы классу, но они не могут добавить новый назначенный инициализатор или деинициализатор классу. Назначенные инициализаторы и деинициализаторы должны всегда предоставляться реализацией исходного класса.

Если вы используете расширения для того, чтобы добавить инициализатор к типу значений, который обеспечивает значения по умолчанию для всех своих хранимых свойств и не определяет какого-либо пользовательского инициализатора, то вы можете вызвать дефолтный инициализатор и почленный инициализатор для того типа значений изнутри инициализатора вашего расширения. Это не будет работать, если вы уже написали инициализатор как часть исходной реализации значения типа, подробнее см. Делегирование инициализатора для типов значения.

Если вы используете расширение для добавления инициализатора в структуру, которая была объявлена в другом модуле, новый инициализатор не может получить доступ к себе до тех пор, пока он не вызовет инициализатор из модуля определения.

Пример ниже определяет структуру Rect для отображения геометрического прямоугольника. Пример так же определяет две вспомогательные структуры Size и Point, обе из которых предоставляют значения по умолчанию 0.0 для всех своих свойств:

struct Size {

var width = 0.0

var height = 0.0

}

struct Point {

var x = 0.0

var y = 0.0

}

struct Rect {

var origin = Point()

var size = Size()

}

Из-за того, что структура Rect предоставляет значения по умолчанию для всех своих свойств, она автоматически получает инициализатор по умолчанию и почленный инициализатор, что описано в главе Дефолтные инициализаторы. Эти инициализаторы могут быть использованы для создания экземпляров Rect:

let defaultRect = Rect()

let memberwiseRect = Rect(origin: Point(x: 2.0, y: 2.0),

size: Size(width: 5.0, height: 5.0))

Вы можете расширить структуру Rect для предоставления дополнительного инициализатора, который принимает определенную точку и размер:

extension Rect {

init(center: Point, size: Size) {

let originX = center.x - (size.width / 2)

let originY = center.y - (size.height / 2)

self.init(origin: Point(x: originX, y: originY), size: size)

}

}

Этот новый инициализатор начинается с вычисления исходной точки, основываясь на значениях свойств center и size. Потом инициализатор вызывает почленный инициализатор структуры init(origin:size:), который хранит новую исходную точку и размеры в соответствующих свойствах:

let centerRect = Rect(center: Point(x: 4.0, y: 4.0),

size: Size(width: 3.0, height: 3.0))

// исходная точка centerRect (2.5, 2.5) и его размер (3.0, 3.0)

Методы в расширениях

Расширения могут добавить новые методы экземпляра или методы типа к уже существующим типам. Следующий пример добавляет новый метод экземпляра repetitions к типу Int:

extension Color {

func repetitions() -> UIColor {

let color = UIColor(red: 1.0, green: 0.3, blue: 0.7, alfa: 1.0)

return color

}

}

Метод repetitions(task:) принимает единственный аргумент типа () -> Void, который указывает на функцию, которая не принимает ни одного параметра и которая не возвращает значения.

После определения расширения вы можете вызвать метод repetitions(task:) на любом целом числе, чтобы выполнить определенное задание целое число раз:

3.repetitions {

print("Hello!")

}

// Hello!

// Hello!

// Hello!

Изменяющиеся методы экземпляра

Методы экземпляров, добавленные в расширении так же могут менять и сам экземпляр. Методы структуры и перечисления, которые изменяют self или его свойства, должны быть отмечены как mutating.

Пример ниже добавляет новый изменяющийся (mutating) метод square для типа Int, который возводит в квадрат исходное значение:

extension Int {

mutating func square() {

self = self \* self

}

}

var someInt = 3

someInt.square()

// теперь переменная someInt имеет значение 9

Сабскрипты в расширениях

Расширения могут добавить новые сабскрипты к существующему типу. Этот пример добавляет сабскрипт целого числа во встроенный тип Int языка Swift. Этот сабскрипт [n] возвращает цифру, которая стоит на n позиции справа:

123456789[0] возвращает 9

123456789[1] возвращает 8

и так далее:

extension Int {

subscript(digitIndex: Int) -> Int {

var decimalBase = 1

for \_ in 0..<digitIndex {

decimalBase \*= 10

}

return (self / decimalBase) % 10

}

}

746381295[0]

// возвращает 5

Вложенные типы в расширениях

Расширения могут добавлять новые вложенные типы к существующим классам, структурам и перечислениям:

extension Int {

enum Kind {

case negative, zero, positive

}

var kind: Kind {

switch self {

case 0:

return .zero

case let x where x > 0:

return .positive

default:

return .negative

}

}

}

**Protocol (протоколы)**

Протокол определяет образцы 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.

В дополнение к определенным требованиям, которые должны быть реализованы подписанными под протокол типами, вы можете расширить протокол, чтобы реализовать некоторые из этих требований или для того, чтобы реализовать дополнительную функциональность, которую смогут использовать подписанные под протокол типы.

1 Протоколы обеспечивают множественное наследование

2 Протоколы не могут хранить состояние

3 Протоколы могут быть унаследованы другими протоколами

4 Протоколы могут применяться к структурам (struct), классам(class) и перечислениям (enum), определяя функционал типов

5 Дженерик протоколы (Generic-protocol) позволяют задавать сложные зависимости между типами и протоколами во время их наследования

6 Протоколы не определяют «сильные» и «слабые» ссылки на переменные

7 В расширениях к протоколам можно описывать конкретные реализации методов, и вычисляемых переменных (computed values)

8 Классовые протоколы разрешают себя наследовать только классам

Синтаксис протокола

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

protocol SomeProtocol {

// определение протокола…

}

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…

}

Требование свойств

Протокол требует у соответствующего ему типа предоставить свойство экземпляра или свойство типа, и это свойство должно иметь конкретное имя и тип. Протокол не уточняет какое должно быть свойство, хранимое или вычисляемое, только лишь указывает на требование имени свойства и типа. Протокол также уточняет должно ли свойство быть доступным, или оно должно быть доступным и устанавливаемым.

Если протокол требует от свойства быть доступным и устанавливаемым, то это требование не может полностью быть удовлетворено константой или вычисляемым свойством только для чтения (read only). Если протокол только требует от свойства читаемости (get), то такое требование может быть удовлетворено любым свойством, и это так же справедливо для устанавливаемого свойства, если это необходимо в вашем коде.

Требуемые свойства всегда объявляются как переменные свойства, с префиксом var. Свойства, значения которых вы можете получить или изменить маркируются { get set } после объявления типа свойства, а свойства, значения которых мы можем только получить, но не изменить { get }.

protocol SomeProtocol {

var mustBeSettable: Int { get set }

var doesNotNeedToBeSettable: Int { get }

}

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 }

}

Требование методов

Протоколы могут требовать реализацию определенных методов экземпляра и методов типа, соответствующими типами протоколу. Эти методы написаны как часть определения протокола в точности и в такой же форме как и методы экземпляра или типа, но только в них отсутствуют фигурные скобки или тело метода целиком. Вариативные параметры допускаются точно так же как и в обычных методах. Дефолтные значения, однако, не могут быть указаны для параметров метода внутри определения протокола.

Как и в случае с требованиями свойств типа, вы всегда указываете префикс static для метода типа. И это верно даже тогда, когда требования к методу типа имеет префикс static или class, когда реализуется классом:

protocol SomeProtocol {

static func someTypeMethod()

}

Требования изменяющих методов

Иногда необходимо для метода изменить (или мутировать) экземпляр, которому он принадлежит. Для методов экземпляра типа значения (структура, перечисление) вы располагаете ключевое слово mutating до слова метода func, для индикации того, что этому методу разрешено менять экземпляр, которому он принадлежит, и/или любое свойство этого экземпляра.

Если вы определяете требуемый протоколом метод экземпляра, который предназначен менять экземпляры любого типа, которые принимают протокол, то поставьте ключевое слово mutating перед именем метода, как часть определения протокола. Это позволяет структурам и перечислениями принимать протокол и удовлетворять требованию метода.

Если вы поставили ключевое слово mutating перед методом требуемым протоколом экземпляра, то вам не нужно писать слово mutating при реализации этого метода для класса. Слово mutating используется только структурами или перечислениями.

Требование инициализатора

Иногда протоколы могут требовать реализацию конкретного инициализатора типами соответствующими протоколу. Вы пишите эти инициализаторы как часть определения протокола, точно так же как и обычные инициализаторы, но только без фигурных скобок или без тела инициализатора:

protocol SomeProtocol {

init(someParameter: Int)

}

Реализация класса соответствующего протоколу с требованием инициализатора

Вы можете реализовать требуемый инициализатор в классе, соответствующем протоколу, в качестве назначенного инициализатора или вспомогательного. В любом случае вам нужно отметить этот инициализатор ключевым словом required:

class SomeClass: SomeProtocol {

required init(someParameter: Int) {

// реализация инициализатора…

}

}

Использование модификатора required гарантирует, что вы проведете явную или унаследованную реализацию требуемого инициализатора на всех подклассах соответствующего класса протоколу, так, чтобы они тоже соответствовали протоколу. Подробнее вы можете прочитать в главе Требуемые инициализаторы.

Вам не нужно обозначать реализацию инициализаторов протокола модификатором required в классах, где стоит модификатор final, потому что конечные классы не могут иметь подклассы. Для более полной информации по модификатору final читайте главу Предотвращение переопределения.

Если подкласс переопределяет назначенный инициализатор суперкласса и так же реализует соответствующий инициализатор протоколу, то обозначьте реализацию инициализатора сразу двумя модификаторами required и override:

protocol SomeProtocol {

init()

}

class SomeSuperClass {

init() {

// реализация инициализатора…

}

}

class SomeSubClass: SomeSuperClass, SomeProtocol {

// "required" от соответсвия протоколу SomeProtocol; "override" от суперкласса SomeSuperClass

required override init() {

// реализация инициализатора…

}

}

Требуемые проваливающиеся инициализаторы

Протоколы могут определять требования проваливающихся инициализаторов для соответствуемых протоколу типов, что определено в главе Проваливающиеся инициализаторы.

Требование проваливающегося инициализатора может быть удовлетворено проваливающимся инициализатором или непроваливающимся инициализатором соответствующего протоколу типа. Требование непроваливающегося инициализатора может быть удовлетворено непроваливающимся инициализатором или неявно развернутым проваливающимся инициализатором.

Протоколы как типы

Протоколы сами по себе не несут какой-то новой функциональности. Тем не менее любой протокол, который вы создаете становится полноправным типом, который вы можете использовать в вашем коде.

Так как это тип, то вы можете использовать протокол во многих местах, где можно использовать другие типы:

Как тип параметра или возвращаемый тип в функции, методе, инициализаторе

Как тип константы, переменной или свойства

Как тип элементов массива, словаря или другого контейнера

Заметка

Из-за того что протоколы являются типами, то их имена начинаются с заглавной буквы для соответствия имен с другими типами Swift (Int, String, Bool, Double…)

Добавление реализации протокола через расширение

Вы можете расширить существующий тип для того, чтобы он соответствовал протоколу, даже если у вас нет доступа к источнику кода для существующего типа. Расширения могут добавлять новые свойства, методы и сабскрипты существующему типу, что таким образом может удовлетворить любым требованиями протокола.

Существующие экземпляры типа автоматически принимают и отвечают требованиям протокола, когда опции, необходимые для соответствия добавляются через расширение типа.

Условное соответствие протоколу

Шаблонный тип может удовлетворять требованиям протокола только при определенных условиях, например, когда общий параметр типа соответствует протоколу. Вы можете сделать общий тип условно соответствующим протоколу, указав ограничения при расширении типа. Напишите эти констрейнты после имени протокола, который вы используете, написав оговорку where. Дополнительные сведения о оговорках where см. В разделе Оговорки where.

Следующее расширение делает экземпляры Array совместимыми с TextRepresentable протоколом всякий раз, когда они хранят элементы типа, которые соответствуют TextRepresentable:

extension Array: TextRepresentable where Element: TextRepresentable {

}

Коллекции типов протокола

Протоколы могут использовать в качестве типов, которые хранятся в таких коллекциях как массивы или словари, что упоминалось ранее в Протоколы как типы.

Наследование протокола

Протокол может наследовать один или более других протоколов и может добавлять требования поверх тех требований протоколов, которые он наследует. Синтаксис наследования протокола аналогичен синтаксису наследования класса, но с возможностью наследовать сразу несколько протоколов, которые разделяются между собой запятыми:

protocol InheritingProtocol: SomeProtocol, AnotherProtocol {

// определение протокола…

}

Классовые протоколы

Вы можете ограничить протокол так, чтобы его могли принимать только классы (но не структуры или перечисления), добавив AnyObject протокол к списку реализации протоколов.

protocol SomeClassOnlyProtocol: AnyObject, SomeInheritedProtocol {

// определение протокола типа class-only

}

В примере выше SomeClassOnlyProtocol может быть принят только классом. Если вы попытаетесь принять протокол SomeClassOnlyProtocol структурой или перечислением, то получите ошибку компиляции.

Используйте протоколы class-only, когда поведение, определяемое протоколом, предполагает или требует, что соответствующий протоколу тип должен быть ссылочного типа, а не типом значении. Для более детального исследования с вашей стороны прочитайте главы: Структуры и перечисления - типы значения и Классы - ссылочный тип.

Композиция протоколов

Иногда бывает удобно требовать тип соответствовать нескольким протоколам за раз. Вы можете скомбинировать несколько протоколов в одно единственное требование при помощи композиции протоколов. Композиции протоколов ведут себя так, как будто вы определили временный локальный протокол, который имеет комбинированные требования ко всем протоколам в композиции. Композиции протоколов не определяют новых типов протоколов.

Композиции протоколов имеют форму SomeProtocol & AnotherProtocol. Вы можете перечислить столько протоколов, сколько нужно, разделяя их между собой знаком амперсанда (&). В дополнение к списку протоколов, композиция протокола также может содержать один тип класса, который можно использовать для указания требуемого суперкласса.

Ниже приведен пример, который комбинирует два протокола Named и Aged в одно единственное требование композиции протоколов в качестве параметра функции:

protocol Named {

var name: String { get }

}

protocol Aged {

var age: Int { get }

}

struct Person: Named, Aged {

var name: String

var age: Int

}

Проверка соответствия протоколу

Вы можете использовать операторы is и as для проверки соответствия протоколу и приведению к определенному протоколу. Приведение к протоколу проходит точно так же как и приведение к типу:

Оператор is возвращает значение true, если экземпляр соответствует протоколу и возвращает false, если нет.

Опциональная версия оператора понижающего приведения as? возвращает опциональное значение типа протокола, и это значение равно nil, если оно не соответствует протоколу.

Принудительная версия оператора понижающего приведения as осуществляет принудительное понижающее приведение, и если оно не завершается успешно, то выскакивает runtime ошибка.

Опциональные требования протокола

Вы можете определить опциональные требования для протокола. Эти требования не обязательно должны быть реализованы для соответствия протоколу. Опциональные требования должны иметь префиксный модификатор optional в качестве части определения протокола. Таким образом вы можете писать код, который взаимодействует с кодом на Objective-C. Имеется в виду, что без @objc код не будет компилироваться, и при этом наличие @objc позволяет коду Swift взаимодействовать с кодом Objective-C. И протокол, и опциональное требование должны иметь атрибут @objc. Обратите внимание, что протоколы с маркировкой @objc могут приниматься только классами, но не структурами или перечислениями.

Когда вы используете опциональное требование свойства или метода, то их тип автоматически становится опциональным. Например, тип метода (Int) -> String становится ((Int) -> String)?. Обратите внимание, что весь тип функции обернут в опциональное значение, а не только возвращаемое значение функции.

Опциональное требование протокола может быть вызвано при помощи опциональной цепочки, чтобы учесть возможность того, что требование не будет реализовано типом, который соответствует протоколу. Вы проверяете реализацию опционального метода, написав вопросительный знак после имени метода, когда он вызывается, например someOptionalMethod?(someArgument). Для более полной информации о опциональной последовательности читайте Опциональная последовательность.

Следующий пример определяет класс Counter, который использует источник внешних данных для предоставления значение их инкремента. Этот источник внешних данных определен протоколом CounterDataSource, который имеет два опциональных требования:

@objc protocol CounterDataSource {

@objc optional func increment(forCount count: Int) -> Int

@objc optional var fixedIncrement: Int { get }

}

Протокол CounterDataSource определяет опциональное требование метода increment(forCount:) и опциональное требование свойства fixedIncrement. Эти требования определяют два разных способа для источника данных для предоставления подходящего значения инкремента для экземпляра Counter.

Строго говоря, вы можете написать пользовательский класс, который соответствует протоколу CounterDataSource без реализации какого-либо требования этого протокола. Они оба опциональные, в конце концов. Хотя технически это допускается, но это не будет реализовываться для хорошего источника данных.

Расширения протоколов

Протоколы могут быть расширены для обеспечения метода и реализации свойства соответствующими типами. Это позволяет вам самостоятельно определить поведение по протоколам, а не по индивидуальному соответствию каждого типа или глобальной функции.

Например, RandomNumberGenerator протокол может быть расширен для обеспечения randomBool() метода, который использует результат вызванного random() метода для возврата random (случайного) значения Bool:

extension AnyProtocol {

}

Создавая расширение по протоколу, все соответствующие типы автоматически получают эту реализацию метода без каких-либо дополнительных изменений.

Расширения протоколов могут добавлять реализацию к соответствующим типам данных, но не могут расширить протокол или унаследовать от другого протокола. Наследование протокола всегда указывается в самом объявлении протокола.