Лабораторная работа №4

Тема занятия: «Использование языка программирования Swift: наследование, протоколы»

Цель: Выполнить разработку приложения с использованием языка программирования Swift: наследование, протоколы.

Время выполнения: 8 часа.

БГУИР

4.1 НАСЛЕДОВАНИЕ

Класс может *наследовать* методы, свойства и другие характеристики другого класса. Когда один класс наследует у другого класса, то наследующий класс называется *подклассом*, класс у которого наследуют - *суперклассом*. Наследование - фундаментальное поведение, которое отделяет классы от других типов Swift.

Классы в Swift могут вызывать или получать доступ к методам, свойствам, индексам, принадлежащим их суперклассам и могут предоставлять свои собственные переписанные версии этих методов, свойств, индексов для усовершенствования или изменения их поведения.

Классы так же могут добавлять наблюдателей свойств к наследованным свойствам для того, чтобы быть в курсе, когда происходит смена значения свойства. Наблюдатели свойств могут быть добавлены для любого свойства, несмотря на то были ли они изначально определены как хранимые свойства или вычисляемые.

4.2 ОПРЕДЕЛЕНИЕ БАЗОВОГО КЛАССА

Любой класс, который ничего не наследует из другого класса, называется *базовым классом*.

Классы в Swift ничего не наследуют от универсального базового класса. Классы, у которых не указан супер класс (родительский класс), называются базовыми, которые вы можете использовать для строительства других классов.

Пример ниже определяет класс Vehicle. Этот базовый класс определяет хранимое свойство currentSpeed, с начальным значением 0.0 (выведенный тип Double). Значение свойства currentSpeed используется вычисляемым нередактируемым свойством description типа String, для создания описания транспортного средства (экземпляра Vehicle).

Так же класс Vehicle определяет метод makeNoise. Этот метод фактически ничего не делает для базового экземпляра класса Vehicle, но будет настраиваться подклассом класса Vehicle чуть позже:

```
class Vehicle {
  var currentSpeed = 0.0
  var description: String {
    return "движется на скорости \(currentSpeed) миль в
час"
  }
  func makeNoise() {
    //ничего не делаем, так как не каждый транспорт шумит
  }
}
```

Вы создаете новый экземпляр класса Vehicle при помощи синтаксиса инициализатора, который написан как TypeName, за которым идут пустые круглые скобки:

```
let someVehicle = Vehicle()
```

Создав новый экземпляр класса Vehicle, вы можете получить доступ к его свойству description, для вывода на экран описания текущей скорости транспорта:

```
print("Транспорт: \(someVehicle.description)")
//Транспорт: движется на скорости 0.0 миль в час
```

Класс Vehicle определяет обычные характеристики для обычного транспортного средства, но особо мы их использовать не можем. Чтобы сделать класс более полезным, вам нужно усовершенствовать его для описания более специфичных видов транспорта.

4.3 НАСЛЕДОВАНИЕ ПОДКЛАССОМ

Наследование является актом создания нового класса на базе существующего класса (базового класса). Подкласс наследует характеристики от существующего класса, который затем может быть усовершенствован. Вы так же можете добавить новые характеристики подклассу.

Для индикации того, что подкласс имеет суперкласс, просто напишите имя подкласса, затем имя суперкласса и разделите их двоеточием:

```
class SomeSubclass: SomeSuperclass {
  // определение подкласса проводится тут
}
```

Приведенный пример определяет подкласс Bicycle с суперклассом Vehicle:

```
class Bicycle: Vehicle {
  var hasBasket = false
}
```

Новый класс Bicycle автоматически собирает все характеристики Vehicle, например, такие свойства как currentSpeed и description и метод makeNoise().

В дополнение к характеристикам, которые он наследует, класс Bicycle определяет свое новое хранимое свойство hasBasket, со значением по умолчанию false (тип свойства выведен как Bool).

БГУИР

По умолчанию, любой новый экземпляр Bicycle, который вы создадите не будет иметь корзину (hasBasket = false). Вы можете установить hasBasket на значение true для конкретного экземпляра Bicycle, после того как он создан:

```
let bicycle = Bicycle()
bicycle.hasBasket = true
```

Вы так же можете изменить унаследованное свойство currentSpeed экземпляра Bicycle и запросить его свойство description:

```
bicycle.currentSpeed = 15.0
print("Велосипед: \(bicycle.description)")
//Велосипед: движется на скорости 15.0 миль в час
```

Подклассы сами могут создавать подклассы. В следующем примере класс Bicycle создает подкласс для двухместного велосипеда известного как «тандем»:

```
class Tandem: Bicycle {
  var currentNumberOfPassengers = 0
}
```

Класс Tandem наследует все свойства и методы Bicycle, который в свою очередь наследует все свойства и методы от Vehicle. Подкласс Tandem так же добавляет новое хранимое свойство currentNumberOfPassengers, которое по умолчанию равно 0.

Если вы создадите экземпляр Tandem, то вы можете работать с любым из его новых и унаследованных свойств. Свойство description, которое является свойством только для чтения, он наследует от Vehicle:

```
let tandem = Tandem()
tandem.hasBasket = true
tandem.currentNumberOfPassengers = 2
tandem.currentSpeed = 22.0
print("Тандем: \(tandem.description)")
// Тандем: движется на скорости 22.0 миль в час
```

4.4 ПЕРЕОПРЕДЕЛЕНИЕ

Подклассы могут проводить свои собственные реализации методов экземпляра, методов класса, свойств экземпляра, свойств класса или индекса, который в противном случае будет наследовать от суперкласса. Это известно как *переопределение*.

Для переопределения характеристик, которые все равно будут унаследованы, вы приписываете к переписываемому определению ключевое слово override. Делая так, вы показываете свое намерение провести переопределение, и что оно будет сделано не по ошибке. Переписывание по

БГУИР

случайности может вызвать непредвиденное поведение, и любое переопределение без ключевого слова override, будет считаться ошибкой при компиляции кода.

Ключевое слово override так же подсказывает компилятору Swift проверить, что вы переопределяете суперкласс класса (или один из его параметров), который содержит то определение, которое вы хотите переопределить. Эта проверка гарантирует, что ваше переопределение корректно.

4.4.1 Доступ к методам, свойствам, индексам суперкласса

Когда вы проводите переопределение метода, свойства, индекса для подкласса, иногда бывает полезно использовать существующую реализацию суперкласса как часть вашего переопределения. Для примера, вы можете усовершенствовать поведение существующей реализации или сохранить измененное значение в существующей унаследованной переменной.

Там, где это уместно, вы можете получить доступ к методу, свойству, индексу версии суперкласса, если будете использовать префикс super:

Переопределенный метод someMethod может вызвать версию суперкласса метода someMethod, написав super.someMethod() внутри переопределения реализации метода.

Переопределённое свойство someProperty может получить доступ к свойству версии суперкласса someProperty как super.someProperty внутри переопределения реализации геттера или сеттера.

Переопределенный индекс для someIndex может получить доступ к версии суперкласса того же индекса как super[someIndex] изнутри переопределения реализации индекса.

4.4.2 Переопределение методов

Вы можете переопределить унаследованный метод экземпляра или класса, чтобы обеспечить индивидуальную или альтернативную версию реализации метода в подклассе.

Следующий пример определяет новый подкласс Train класса Vehicle, который переопределяет метод makeNoise(), который Train наследует от Vehicle:

```
class Train: Vehicle {
  override func makeNoise() {
    print("Чу-чу")
  }
}
```

Если вы создаете новый экземпляр класса Train и вызовите его метод makeNoise(), вы увидите, что версия метода подкласса Train вызывается вот так:

```
let train = Train()
train.makeNoise()
// Выведет "Чу-чу"
```

4.4.3 Переопределение свойств

Вы можете переопределить унаследованные свойства класса или экземпляра для установки вашего собственного геттера и сеттера для этого свойства, или добавить наблюдателя свойства для наблюдения за переопределяемым свойством, когда меняется лежащее в основе значение свойства.

4.4.4 Переопределения геттеров и сеттеров свойства

Вы можете предусмотреть пользовательский геттер (и сеттер, если есть в этом необходимость) для переопределения любого унаследованного свойства, несмотря на то, как свойство было определено в самом источнике, как свойство хранения или как вычисляемое. Подкласс не знает каким является унаследованное свойство хранимым или вычисляемым, все что он знает, так это имя свойства и его тип. Вы всегда должны констатировать и имя, и тип свойства, которое вы переопределяете, для того чтобы компилятор мог проверить соответствие и наличие переопределяемого свойства у суперкласса.

Вы можете представить унаследованное свойство только для чтения, как свойство, которое можно читать и редактировать, прописывая и геттер и сеттер в вашем переопределяемом свойстве подкласса. Однако вы не можете сделать наоборот, то есть сделать свойство редактируемое и читаемое только свойством для чтения.

Если вы предоставляете сеттер как часть переопределения свойства, то вы должны предоставить и геттер для этого переопределения. Если вы не хотите изменять значение наследуемого свойства внутри переопределяемого геттера, то вы можете просто передать через наследуемое значение, возвращая super.someProperty от геттера, где someProperty - имя параметра, который вы переопределяете.

Следующий пример определяет класс Car, который является подклассом Vehicle. Класс Car предоставляет новое свойство хранения gear, имеющее значение по умолчанию равное 1. Класс Car так же переопределяет свойство description, которое он унаследовал от Vehicle, для предоставления собственного описания, которое включает в себя текущую передачу:

```
class Car: Vehicle {
  var gear = 1
  override var description: String {
    return super.description + " на передаче \(gear)"
  }
}
```

Переопределение свойства description начинается с super.description, который возвращает свойство description класса Vehicle. Версия класса Car свойства description добавляет дополнительный текст в конец описания текущего свойства description.

Если вы создадите экземпляр класса Car и зададите свойства gear, currentSpeed, то вы увидите что его свойство description возвращает новое описание класса Car:

```
let car = Car()
car.currentSpeed = 25.0
car.gear = 3
print("Машина: \(car.description)")
// Выведет "Машина: движется на скорости 25.0 миль в час на передаче 3"
```

4.4.5 Переопределение наблюдателей свойства

Вы можете использовать переопределение свойства для добавления наблюдателей к унаследованному свойству. Это позволяет вам получать уведомления об изменении значения унаследованного свойства, несмотря на то, как изначально это свойство было реализовано. Для большей информации о наблюдателях свойств читайте в главе Наблюдатели свойств.

Вы не можете добавить наблюдателей свойства на унаследованное константное свойство или на унаследованные вычисляемые свойства только для чтения. Значение этих свойств не может меняться, так что нет никакого смысла вписывать willSet, didSet как часть их реализации.

Также заметим, что вы не можете обеспечить одно и то же свойство и переопределяемым наблюдателем, и переопределяемым сеттером. Если вы хотите наблюдать за изменениями значения свойства, и вы готовы предоставить пользовательский сеттер для этого свойства, то вы можете просто наблюдать за изменением какого-либо значения из сеттера.

В следующем примере определим новый класс AutomaticCar, который является подклассом Car. Класс AutomaticCar представляет машину с автоматической коробкой передач, которая автоматически переключает передачи в зависимости от текущей скорости:

```
class AutomaticCar: Car {
  override var currentSpeed: Double {
    didSet {
      gear = Int(currentSpeed / 10.0) + 1
    }
  }
}
```

Куда бы вы не поставили свойство currentSpeed экземпляра класса AutomaticCar, наблюдатель didSet свойства устанавливает свойство

экземпляра gear в подходящее значение передачи в зависимости от скорости. Если быть точным, то наблюдатель свойства выбирает передачу как значение равное currentSpeed поделенная на 10 и округленная вниз и выбираем ближайшее целое число + 1. Если скорость равна 10.0, то передача равна 2, если скорость 35.0, то передача 4:

```
let automatic = AutomaticCar()
automatic.currentSpeed = 35.0
print("Машина с автоматом: \(automatic.description)")
//Выведет "Машина с автоматом: движется на скорости 35.0 миль в час на передаче 4"
```

4.5 ПРЕДОТВРАЩЕНИЕ ПЕРЕОПРЕДЕЛЕНИЙ

Вы можете предотвратить переопределение метода, свойства или индекса, обозначив его как *конечный*. Сделать это можно написав ключевое слово final перед ключевым словом метода, свойства или индекса (final var, final func, final class func, и final subscript).

Любая попытка переписать конечный метод, свойство или индекс в подклассе приведет к ошибке компиляции. Методы, свойства и индексы, которые вы добавляете в класс в расширении, так же могут быть отмечены как конечные внутри определения расширения.

Вы можете отметить целый класс как конечный или финальный, написав слово final перед ключевым словом class (final class). Любая попытка унаследовать класс также приведет к ошибке компиляции.

4.6 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.

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.

4.7 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…
}
```

4.8 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 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 }
}
```

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"
```

Этот пример определяет структуру Person, которая отображает персону с конкретным именем. Она утверждает, что она принимает протокол FullyNamed в качестве первой строки собственного определения.

Каждый экземпляр Person имеет единственное свойство fullName типа String. Это удовлетворяет единственному требованию протокола FullyNamed, и это значит, что Person корректно соответствует протоколу. (Swift сообщает об ошибке во время компиляции, если требования протокола выполняются не полностью.)

Ниже представлен более сложный класс, который так же принимает и соответствует протоколу FullyNamed:

```
class Starship: FullyNamed {
    var prefix: String?
    var name: String
```

```
    init(name: String, prefix: String? = nil) {
        self.name = name
        self.prefix = prefix
    }
    var fullName: String {
        return (prefix != nil ? prefix! + " " : "") + name
    }
}
var ncc1701 = Starship(name: "Enterprise", prefix: "USS")
// ncc1701.fullName равен "USS Enterprise"
```

Класс реализует требуемое свойство fullName, в качестве вычисляемого свойства только для чтения (для космического корабля). Каждый экземпляр класса Starship хранит обязательный name и опциональный prefix. Свойство fullName использует значение prefix, если оно существует и устанавливает его в начало name, чтобы получилось целое имя для космического корабля.

4.9 ТРЕБОВАНИЕ МЕТОДОВ

Протоколы могут требовать реализацию определенных методов экземпляра и методов типа, соответствующими типами протоколу. Эти методы написаны как часть определения протокола в точности 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()
}
```

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.

Ниже приведена реализация класса, который принимает и соответствует протоколу RandomNumberGenerator. Этот класс реализует алгоритм генератора псевдослучайных чисел, известный как алгоритм *линейного конгруэнтного генератора*:

```
class LinearCongruentialGenerator: RandomNumberGenerator {
    var lastRandom = 42.0
    let m = 139968.0
    let a = 3877.0
    let c = 29573.0
    func random() -> Double {
        lastRandom = ((lastRandom * a + 
c).truncatingRemainder(dividingBy:m))
        return lastRandom / m
    }
}
let generator = LinearCongruentialGenerator()
print("Here's a random number: \(generator.random())")
// Выведет "Случайное число: 0.37464991998171"
print("And another one: \(generator.random())")
// Выведет "Другое случайное число: 0.729023776863283"
```

4.10 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.

Пример ниже определяет протокол Togglable, который определяет единственный требуемый метод экземпляра toggle(). Как и предполагает имя метода, он переключает или инвертирует состояние любого типа, обычно меняя свойство этого типа.

БГУИР

Метод toggle() имеет слово mutating как часть определения протокола Togglable, для отображения того, что этот метод меняет состояние соответствующего протоколу экземпляра при своем вызове:

```
protocol Togglable {
    mutating func toggle()
}
```

Если вы реализуете протокол Togglable для структур или перечислений, то эта структура или перечисление может соответствовать протоколу предоставляя реализацию метода toggle(), который так же будет отмечен словом mutating.

Пример ниже определяет перечисление OnOffSwitch. Это перечисление переключается между двумя состояниями, отмеченными двумя случаями перечислениям .on и .off. Реализация метода toggle перечисления отмечена словом mutating, чтобы соответствовать требованию протокола:

```
enum OnOffSwitch: Togglable {
    case off, on
    mutating func toggle() {
        switch self {
        case .off:
            self = .on
        case .on:
            self = .off
        }
    }
}
var lightSwitch = OnOffSwitch.off
lightSwitch.toggle()
// lightSwitch теперь равен .on
```

4.11 ТРЕБОВАНИЕ ИНИЦИАЛИЗАТОРА

Иногда протоколы могут требовать реализацию конкретного инициализатора типами соответствующими протоколу. Вы пишете 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)
}
```

4.12.1 Реализация класса соответствующего протоколу с требованием инициализатора

БГУИР

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.

Вам не нужно обозначать реализацию инициализаторов протокола модификатором required в классах, где стоит модификатор final, потому что конечные классы не могут иметь подклассы.

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

```
protocol SomeProtocol {
    init()
}

class SomeSuperClass {
    init() {
        // реализация инициализатора…
    }
}

class SomeSubClass: SomeSuperClass, SomeProtocol {
    // "required" от соответствия протоколу SomeProtocol;
"override" от суперкласса SomeSuperClass
    required override init() {
        // реализация инициализатора…
    }
}
```

4.12 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

БГУИР

как тип константы, переменной или свойства

как тип элементов массива, словаря или другого контейнера

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

Вот пример использования протокола в качестве типа:

```
class Dice {
    let sides: Int
    let generator: RandomNumberGenerator
    init(sides: Int, generator: RandomNumberGenerator) {
        self.sides = sides
        self.generator = generator
    }
    func roll() -> Int {
        return Int(generator.random() * Double(sides)) + 1
    }
}
```

Этот пример определяет новый класс Dice, который отображает игральную кость с n количеством сторон для настольной игры. Экземпляры Dice имеют свойство sides, которое отображает количество сторон, которое они имеют, так же кубики имеют свойство generator, которое предоставляет генератор случайных чисел, из которого и берутся значения броска игрального кубика.

Свойство generator имеет тип RandomNumberGenerator, таким образом вы можете использовать в этом свойстве экземпляр любого типа, соответствующий протоколу RandomNumberGenerator. Больше ничего не требуется от экземпляра, присваемого этому свойству, кроме того, что этот экземпляр должен принимать протокол RandomNumberGenerator.

Dice так же имеет инициализатор для установки начальных значений. Этот инициализатор имеет параметр generator, который так же является типом RandomNumberGenerator. Вы можете передать значение любого соответствующего протоколу типа в этот параметр, когда инициализируете новый экземпляр Dice.

Dice предоставляет один метод экземпляра - roll, который возвращает целое значение от 1 и до количества сторон на игральной кости. Этот метод вызывает генератор метода random(), для создания нового случайного числа от 0.0 и до 1.0, а затем использует это случайное число для создания значения броска игральной кости в соответствующем диапазоне (1…n). Так как мы знаем, что generator принимает RandomNumberGenerator, то это гарантирует нам, что у нас будет метод random().

Вот как используется класс Dice для создания шестигранной игральной кости с экземпляром LinearCongruentialGenerator в качестве генератора случайных чисел:

ЧЕРНОВИК

```
var d6 = Dice(sides: 6, generator:
LinearCongruentialGenerator())
for _ in 1...5 {
    print("Бросок игральной кости равен \(d6.roll())")
}
// Бросок игральной кости равен 3
// Бросок игральной кости равен 5
// Бросок игральной кости равен 4
// Бросок игральной кости равен 5
// Бросок игральной кости равен 4
```

4.13 ДЕЛЕГИРОВАНИЕ

Делегирование – это шаблон, который позволяет классу или структуре передавать (или делегировать) некоторую ответственность экземпляру другого типа. Этот шаблон реализуется определением протокола, который инкапсулирует делегируемые полномочия, таким образом, что соответствующий протоколу тип (делегат) гарантировано получит функциональность, которая была ему делегирована. Делегирование может быть использовано для ответа на конкретное действие или для получения данных из внешнего источника без необходимости знания типа источника.

Пример ниже определяет два протокола для использования в играх, основанных на бросках игральных костей:

```
protocol DiceGame {
    var dice: Dice { get }
    func play()
}
protocol DiceGameDelegate: AnyObject {
    func gameDidStart(_ game: DiceGame)
    func game(_ game: DiceGame, didStartNewTurnWithDiceRoll
diceRoll: Int)
    func gameDidEnd(_ game: DiceGame)
}
```

Протокол DiceGame является протоколом, который может быть принят любой игрой, которая включает игральную кость.

Протокол DiceGameDelegate может быть принят любым типом для отслеживания прогресса DiceGame. Для предотвращения цикла сильных ссылок, делегаты определены как weak ссылки. Маркировка протокола "только для классов" позволяет классу SnakesAndLadders далее в этой главе объявить, что его делегат должен использовать слабую ссылку. Классовые протоколы наследуют протокол AnyObject.

Вот версия игры «Змеи и лестницы», которая адаптирована под использование экземпляра Dice для своих бросков кости, для соответствия протоколу DiceGame и для уведомления DiceGameDelegate о прогрессе:

```
class SnakesAndLadders: DiceGame {
    let finalSquare = 25
```

БУКВ

```
        let dice = Dice(sides: 6, generator:
LinearCongruentialGenerator())
        var square = 0
        var board: [Int]
        init() {
            board = Array(repeating: 0, count: finalSquare + 1)
            board[03] = +08; board[06] = +11; board[09] = +09;
board[10] = +02
            board[14] = -10; board[19] = -11; board[22] = -02;
board[24] = -08
        }
        weak var delegate: DiceGameDelegate?
        func play() {
            square = 0
            delegate?.gameDidStart(self)
            gameLoop: while square != finalSquare {
                let diceRoll = dice.roll()
                delegate?.game(self,
didStartNewTurnWithDiceRoll: diceRoll)
                switch square + diceRoll {
                case finalSquare:
                    break gameLoop
                case let newSquare where newSquare >
finalSquare:
                    continue gameLoop
                default:
                    square += diceRoll
                    square += board[square]
                }
            }
            delegate?.gameDidEnd(self)
        }
    }
```

Эта версия игры обернута в класс SnakesAndLadders, который принимает протокол DiceGame. Он предоставляет свойство dice и метод play() для соответствия протоколу. (Свойство dice объявлено как константное свойство, потому что оно не нуждается в изменении значения после инициализации, а протокол требует только чтобы оно было доступным.)

Настройка игры «*Змеи и лестницы*» происходит в инициализаторе класса init(). Вся логика игры перемещается в метод play протокола, который использует требуемое свойство протокола для предоставления значений броска игральной кости.

Обратите внимание, что свойство delegate определено как *опциональное* DiceGameDelegate, потому что делегат не требуется для игры. Так как оно является опциональным типом, свойство delegate автоматически устанавливает начальное значение равное nil. Таким образом, у каждого экземпляра игры есть установки свойства подходящему делегату. Так как DiceGameDelegate является протоколом только для классов, то мы должны

БГУИР

установить модификатор weak у делегата, чтобы избежать цикла сильных ссылок.

DiceGameDelegate предоставляет три метода для отслеживания прогресса игры. Эти три метода были включены в логику игры внутри метода play(), и вызываются когда начинается новая игра, начинается новый ход или игра кончается.

Так как свойство delegate является опциональным DiceGameDelegate, метод play() использует опциональную последовательность каждый раз, как вызывается этот метод у делегата. Если свойство delegate равно nil, то этот вызов этого метода делегатом проваливается без возникновения ошибки. Если свойство delegate не nil, вызываются методы делегата, которые передаются в экземпляр SnakesAndLadders в качестве параметра.

Следующий пример показывает класс DiceGameTracker, который принимает протокол DiceGameDelegate:

```
class DiceGameTracker: DiceGameDelegate {
    var numberOfTurns = 0
    func gameDidStart(_ game: DiceGame) {
        numberOfTurns = 0
        if game is SnakesAndLadders {
            print("Начали новую игру Змеи и лестницы")
        }
        print("У игральной кости \(game.dice.sides) граней")
    }
    func game(_ game: DiceGame, didStartNewTurnWithDiceRoll
diceRoll: Int) {
        numberOfTurns += 1
        print("Выкинули \(diceRoll)")
    }
    func gameDidEnd(_ game: DiceGame) {
        print("Длительность игры \(numberOfTurns) хода")
    }
}
```

DiceGameTracker реализует все три метода, которые требует DiceGameDelegate. Он использует эти методы для отслеживания количества ходов, которые были сделаны в игре. Он сбрасывает значение свойства numberOfTurns на ноль, когда начинается игра и увеличивает каждый раз, как начинается новый ход и выводит общее число ходов, как только кончается игра.

Реализация gameDidStart(_:), показанная ранее, использует параметр game для отображения вступительной информации об игре, в которую будут играть. Параметр game имеет тип DiceGame, но не SnakesAndLadders, так что gameDidStart(_:) может получить и использовать только те методы и свойства, которые реализованы как часть протокола DiceGame. Однако метод все еще может использовать приведение типов для обращения к типу основного (исходного) экземпляра. В этом примере, он проверяет

БГУИР

действительно ли game является экземпляром SnakesAndLadders или нет, а потом выводит соответствующее сообщение.

gameDidStart(_:) так же получает доступ к свойству dice, передаваемого параметра game. Так как известно, что game соответствует протоколу DiceGame, то это гарантирует наличие свойства dice, таким образом метод gameDidStart(_:) может получить доступ и вывести сообщение о свойстве кости sides, независимо от типа игры, в которую играют.

Теперь давайте взглянем на то, как выглядит DiceGameTracker в действии:

```
let tracker = DiceGameTracker()
let game = SnakesAndLadders()
game.delegate = tracker
game.play()
// Начали новую игру Змеи и лестницы
// У игральной кости 6 граней
// Выкинули 3
// Выкинули 5
// Выкинули 4
// Выкинули 5
// Длительность игры 4 хода
```

4.14 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.

К примеру, этот протокол TextRepresentable может быть реализован любым типом, который может отображать текст. Это может быть собственное описание или текстовая версия текущего состояния:

```
protocol TextRepresentable {
    var textualDescription: String { get }
}
```

Класс Dice, о котором мы говорили ранее, может быть расширен для принятия и соответствия протоколу TextRepresentable:

```
extension Dice: TextRepresentable {
    var textualDescription: String {
        return "Игральная кость с \(sides) гранями"
```

```
        }
    }
```

Это расширение принимает новый протокол в точности так же, как если бы Dice был представлен внутри его первоначальной реализации. Имя протокола предоставляется после имени типа и отделяется от имени двоеточием, и реализация всех требований протокола обеспечивается внутри фигурных скобок расширения.

Теперь экземпляр Dice может быть использован как TextRepresentable:

```
    let d12 = Dice(sides: 12, generator: LinearCongruentialGenerator())
    print(d12.textualDescription)
    // Выведет "Игральная кость с 12 гранями"
```

Аналогично игровой класс SnakesAndLadders может быть расширен для того, чтобы смог принять и соответствовать протоколу TextRepresentable:

```
    extension SnakesAndLadders: TextRepresentable {
        var textualDescription: String {
            return "Игра Змеи и Лестницы с полем в \(finalSquare) клеток"
        }
    }
    print(game.textualDescription)
    // Выведет "Игра Змеи и Лестницы с полем в 25 клеток"
```

4.14.1 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 ограничения после имени протокола, который вы используете, написав оговорку 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 {
        var textualDescription: String {
            let itemsAsText = self.map { $0.textualDescription }
            return "[" + itemsAsText.joined(separator: ", ") + "]"
        }
    }
```

```
let myDice = [d6, d12]
print(myDice.textualDescription)
// Prints "[A 6-sided dice, A 12-sided dice]"
```

4.14.2 Принятие протокола через расширение

Если тип уже соответствует всем требованиям протокола, но еще не заявил, что он принимает этот протокол, то вы можете сделать это через пустое расширение:

```
struct Hamster {
    var name: String
    var textualDescription: String {
        return "Хомяка назвали \(name)"
    }
}
extension Hamster: TextRepresentable {}
```

Экземпляры Hamster теперь могут быть использованы в тех случаях, когда нужен тип TextRepresentable:

```
let simonTheHamster = Hamster(name: "Фруша")
let somethingTextRepresentable: TextRepresentable = simonTheHamster
print(somethingTextRepresentable.textualDescription)
// Выведет "Хомяка назвали Фруша"
```

Типы не принимают протоколы автоматически, если они удовлетворяют их требованиям. Принятие протокола должно быть объявлено в явной форме.

4.15 ПРИНЯТИЕ ПРОТОКОЛА ЧЕРЕЗ СИНТЕЗИРОВАННУЮ РЕАЛИЗАЦИЮ

Swift может автоматически предоставлять соответствие таких протоколов как Equatable, Hashable и Comparable в большинстве простых случаев. Использование синтезированной реализации означает для нас, что мы не должны будем писать повторяющийся шаблонный код, для того, чтобы реализовать требования протокола.

Swift предоставляет синтезированную реализацию протокола Equatable для следующих кастомных типов:

Структуры, которые имеют только свойства хранения и соответствуют протоколу Equatable

Перечисления, которые имеют только ассоциативные типы и соответствуют протоколу Equatable

Перечисления, которые не имеют ассоциативных типов

ГГУИР

Чтобы получить синтезированную реализацию оператора ==, вам нужно объявить о соответствии протоколу Equatable в файле, который содержит оригинальное объявление без реализации оператора ==. По умолчанию Equatable предоставит свою дефолтную реализацию оператора !=.

В примере ниже представлена структура Vector3D для позиционирования вектора в 3D пространстве, то есть с координатами x, y, z, аналогично тому как это было представлено в структуре Vector2D. Так как все три свойства x, y, z соответствуют протоколу Equatable, то Vector3D получает синтезированную реализацию оператора равенства.

```
struct Vector3D: Equatable {
    var x = 0.0, y = 0.0, z = 0.0
}

let twoThreeFour = Vector3D(x: 2.0, y: 3.0, z: 4.0)
let anotherTwoThreeFour = Vector3D(x: 2.0, y: 3.0, z: 4.0)
if twoThreeFour == anotherTwoThreeFour {
    print("Эти два вектора эквивалентны.")
}
// Выведет "Эти два вектора эквивалентны."
```

Swift предоставляет синтезированную реализацию протокола Hashable для следующих кастомных типов:

Структуры имеют только свойства хранения, которые соответствуют протоколу Hashable

Перечисления, которые имеют только ассоциативные типы, которые соответствуют протоколу Hashable

Перечисления, которые не имеют ассоциативных типов

Для получения синтезированной реализации метода hash(into:), нужно объявить о соответсвии протоколу Hashable в файле, который содержит оригинальное объявление без реализации метода hash(into:).

Swift предоставляет синтезированную реализацию Comparable для перечислений, у которых нет сырого значения (rawValue). Если перечисление имеет ассоциативные типы, то они все должны соответствовать протоколу Comparable. Для получения синтезированной реализации оператора <, объявите о соответствии протоколу Comparable в файле, который содержит оригинальное объявление перечисления, без реализации оператора <. Дефолтная реализация операторов протокола Comparable <=, > и >= предоставляет реализацию остальных операторов сравнения.

Пример ниже определяет перечисление SkillLevel с кейсами beginner, intermediate, expert. Кейс expert дополнительно ранжируется по количеству звезд.

```
enum SkillLevel: Comparable {
    case beginner
    case intermediate
    case expert(stars: Int)
```

БГУИР

```
    }
    var levels = [SkillLevel.intermediate, SkillLevel.beginner,
                  SkillLevel.expert(stars:                  5),
SkillLevel.expert(stars: 3)]
    for level in levels.sorted() {
        print(level)
    }
    // Выведет "beginner"
    // Выведет "intermediate"
    // Выведет "expert(stars: 3)"
    // Выведет "expert(stars: 5)"
```

4.16 КОЛЛЕКЦИИ ТИПОВ ПРОТОКОЛА

Протоколы могут использоваться в качестве типов, которые хранятся в таких коллекциях как массивы или словари.

Пример ниже создает массив из элементов типа TextRepresentable:

```
    let    things:    [TextRepresentable]    =    [game,    d12,
simonTheHamster]
```

Теперь мы можем перебирать элементы массива и выводить текстовое отображение каждого из них:

```
    for thing in things {
        print(thing.textualDescription)
    }
    // Игра Змеи и Лестницы с полем в 25 клеток
    // Игральная кость с 12 гранями
    // Хомяка назвали Фруша
```

Обратите внимание, что константа things является типом TextRepresentable. Она не является типом Dice, или DiceGame, или Hamster, даже в том случае, если базовый тип является одним из них. Тем не менее из-за того, что она типа TextRepresentable, а все что имеет тип TextRepresentable, имеет метод textualDescription, что значит, что можно безопасно вызывать thing.textualDescription каждую итерацию цикла.

4.17 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…
```

БГУИР

```
}
```

Ниже приведен пример протокола, который наследует протокол TextRepresentable, о котором мы говорили ранее:

```
protocol PrettyTextRepresentable: TextRepresentable {
    var prettyTextualDescription: String { get }
}
```

Этот пример определяет новый протокол PrettyTextRepresentable, который наследует из TextRepresentable. Все, что соответствует протоколу PrettyTextRepresentable, должно удовлетворять всем требованиям TextRepresentable, плюс дополнительные требования введенные от протокола PrettyTextRepresentable. В этом примере PrettyTextRepresentable добавляет единственное требование обеспечить read-only свойство экземпляра prettyTextualDescription, которое возвращает String.

Класс SnakesAndLadders может быть расширен, чтобы иметь возможность принять и соответствовать PrettyTextRepresentable:

```
extension SnakesAndLadders: PrettyTextRepresentable {
    var prettyTextualDescription: String {
        var output = textualDescription + ":\n"
        for index in 1...finalSquare {
            switch board[index] {
            case let ladder where ladder > 0:
                output += "▲ "
            case let snake where snake < 0:
                output += "▼ "
            default:
                output += "○ "
            }
        }
        return output
    }
}
```

Это расширение утверждает, что оно принимает протокол PrettyTextRepresentable и реализует свойство prettyTextualDescription для типа SnakesAndLadders. Все, что является типом PrettyTextRepresentable, так же должно быть и TextRepresentable, таким образом, реализация prettyTextualDescription начинается с обращения к свойству textualDescription из протокола TextRepresentable для начала вывода строки. Затем он добавляет двоеточие и символ разрыва строки для начала текстового отображения. Затем он проводит перебор элементов массива (клеток доски) и добавляет их геометрические формы для отображения контента:

1. Если значение клетки больше нуля, то это является началом лестницы, и это отображается символом ▲.

БГУИР

2. Если значение клетки меньше нуля, то это голова змеи, и эта ячейка имеет символ ▼.

3. И наоборот, если значение клетки равно нулю, то это “свободная” клетка, которая отображается символом ○.

Эта реализация метода может быть использована для вывода текстового описания любого экземпляра SnakesAndLadders:

```
print(game.prettyTextualDescription)
// Игра Змеи и Лестницы с полем в 25 клеток:
// ○ ○ ▲ ○ ○ ▲ ○ ○ ▲ ▲ ○ ○ ○ ▼ ○ ○ ○ ○ ▼ ○ ○ ▼ ○ ▼ ○
```

4.18 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 значения.

4.19 КОМПОЗИЦИЯ ПРОТОКОЛОВ

Иногда бывает удобно требовать тип, который будет соответствовать сразу нескольким протоколам. 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 {
```

БГУИР

```
        var name: String { get }
    }
    protocol Aged {
        var age: Int { get }
    }
    struct Person: Named, Aged {
        var name: String
        var age: Int
    }
    func wishHappyBirthday(to celebrator: Named & Aged) {
        print("С Днем Рождения, \(celebrator.name)! Тебе уже
\(celebrator.age)!")
    }
    let birthdayPerson = Person(name: "Сашка", age: 21)
    wishHappyBirthday(to: birthdayPerson)
    // Выведет "С Днем Рождения, Сашка! Тебе уже 21!"
```

В этом примере мы определяем протокол Named с единственным требованием свойства name типа String, значение которого мы можем получить. Так же мы определяем протокол Aged с единственным требованием свойства age типа Int, значение которого мы так же должны иметь возможность получить. Оба этих протокола принимаются структурой Person.

Этот пример определяет функцию wishHappyBirthday(to:), которая принимает единственный параметр celebrator. Тип этого параметра Named & Aged, что означает “любой тип, который соответствует сразу двум протоколам Aged и Named”. Не важно какой тип передается в качестве параметра функции до тех пор, пока он соответствует этим протоколам.

Далее в примере мы создаем экземпляр birthdayPerson класса Person и передаем этот новый экземпляр в функцию wishHappyBirthday(to:). Из-за того что Person соответствует двум протоколам, то функция wishHappyBirthday(to:) может вывести поздравление с днем рождения.

Следующий пример показывает как вы можете объединить протокол Named с классом Location:

```
    class Location {
        var latitude: Double
        var longitude: Double
        init(latitude: Double, longitude: Double) {
            self.latitude = latitude
            self.longitude = longitude
        }
    }
    class City: Location, Named {
        var name: String
        init(name: String, latitude: Double, longitude: Double)
{
            self.name = name
            super.init(latitude: latitude, longitude: longitude)
        }
```

```
}
func beginConcert(in location: Location & Named) {
    print("Hello, \(location.name)!")
}

let seattle = City(name: "Seattle", latitude: 47.6,
longitude: -122.3)
beginConcert(in: seattle)
// Выведет "Hello, Seattle!"
```

Функция beginConcert(in:) принимает параметр типа Location и Named, что означает любой тип, который является подклассом Location, и который будет реализовывать протокол Named. В этом случае City удовлетворяем обоим требованиям.

Передавать birthdayPerson в функцию beginConcert(in:) некорректно, так как Person не является подклассом Location. И наоборот, если вы создали подкласс Location, который не реализует протокол Named, то вызов метода beginConcert(in:) с этим экземпляром так же является некорректным.

4.20 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.

Этот пример определяет протокол HasArea с единственным требованием свойства area типа Double (доступное свойство):

```
protocol HasArea {
    var area: Double { get }
}
```

Ниже представлены два класса Circle, Country, оба из которых соответствуют протоколу HasArea:

```
class Circle: HasArea {
    let pi = 3.1415927
    var radius: Double
    var area: Double { return pi * radius * radius }
    init(radius: Double) { self.radius = radius }
}
```

```
class Country: HasArea {
    var area: Double
    init(area: Double) { self.area = area }
}
```

Класс Circle реализует требование свойства area в качестве вычисляемого свойства, основываясь на хранимом свойстве radius. Класс Country реализует требование area напрямую в качестве хранимого свойства. Оба класса корректно соответствуют протоколу HasArea.

Ниже приведен класс Animal, который не соответствует протоколу HasArea:

```
class Animal {
    var legs: Int
    init(legs: Int) { self.legs = legs }
}
```

Классы Circle, Country, Animal не имеют общего базового класса. Тем не менее они все являются классами, и их экземпляры могут быть использованы для инициализации массива, который хранит значения типов AnyObject:

```
let objects: [AnyObject] = [
    Circle(radius: 2.0),
    Country(area: 243_610),
    Animal(legs: 4)
]
```

Массив objects инициализирован при помощи литерала, содержащего экземпляры Circle, который имеет radius равный 2, экземпляр типа Country, который инициализирован площадью Великобритании в квадратных километрах, и экземпляром класса Animal, который инициализирован количеством ног.

Массив objects может быть перебран, и каждый элемент массива может быть проверен на соответствие протоколу HasArea:

```
for object in objects {
    if let objectWithArea = object as? HasArea {
        print("Площадь равна \(objectWithArea.area)")
    } else {
        print("Что-то такое, что не имеет площади")
    }
}
// Площадь равна 12.5663708
// Площадь равна 243610.0
// Что-то такое, что не имеет площади
```

БГУИР

Каждый раз, когда объект массива соответствует протоколу HasArea, возвращается опциональное значение при помощи оператора as?, которое разворачивается при помощи опциональной связки в константу objectWithArea. Константа objectWithArea является типом HasArea, таким образом, свойство area может быть доступно и выведено на экран способом вывода через сам тип.

Обратите внимание, что базовые объекты не меняются в процессе приведения типа. Они остаются Circle, Country и Animal. Однако в момент, когда они хранятся в константе objectWithArea, известно лишь то, что они являются типом HasArea, так что мы можем получить доступ только к свойству area.

4.21 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.

Класс Counter, определенный ниже, имеет опциональное свойство dataSource типа CounterDataSource?:

```
class Counter {
    var count = 0
    var dataSource: CounterDataSource?
    func increment() {
        if let amount = dataSource?.increment?(forCount:
count) {
            count += amount
        } else if let amount = dataSource?.fixedIncrement {
            count += amount
        }
    }
}
```

Класс Counter хранит свое текущее значение в переменном свойстве count. Класс Counter так же определяет метод increment, который увеличивает свойство count, каждый раз, когда вызывается этот метод.

Метод increment сначала пытается получить значение инкремента, заглядывая в реализацию метода increment(forCount:) в его источнике данных. Метод increment использует опциональную последовательность для попытки вызвать increment(forCount:) и передает текущее значение count как единственный аргумент метода.

Обратите внимание, что здесь всего *два* уровня опциональной последовательности. Первый - возможный источник данных dataSource, который может быть nil, так что dataSource имеет вопросительный знак после имени для индикации того, что метод increment(forCount:) может быть вызван только в том случае, если dataSource не nil. Второй уровень говорит нам о том, что даже если dataSource существует, у нас все равно нет гарантии того, что он реализует метод increment(forCount:), потому что это опциональное требование. Есть вероятность, что increment(forCount:) так же не реализован из-за опциональной цепочки. Вызов increment(forCount:) происходит только, если increment(forCount:) существует и не равен nil.

БГУИР

Именно по этой причине increment(forCount:) записан с вопросительным знаком после своего имени.

Так как вызов increment(forCount:) может провалиться по одной из этих двух причин, вызов возвращает нам значение типа опционального Int. Это верно даже если increment(forCount:) определено как возвращающее неопциональное значение Int в определении CounterDataSource. Даже если подряд идут две опциональные операции, одна сразу после другой, то результат все равно будет иметь единственный завернутый опционал.

После вызова increment(forCount:) опциональный Int, который он возвращает, разворачивается в константу amount, при помощи опциональной связки. Если опциональный Int содержит значения, то есть, если делегат и метод существуют, и метод вернул значение, то неразвернутое значение amount прибавляется в свойство count, и на этом реализация завершается.

Если же нет возможности получить значение из метода increment(forCount:) по причине dataSource равен nil или из-за того что у источника данных нет реализации метода increment(forCount:), а следовательно вместо этого метод increment пытается получить значение от источника данных fixedIncrement. Свойство fixedIncrement является опциональным требованием, так что его имя так же написано в опциональной последовательности с вопросительным знаком, что служит индикатором того, что попытка получить значение этого свойства может привести к провалу. Как и раньше, возвращаемое значение является опциональной Int, даже тогда fixedIncrement определен как свойство типа неопционального Int, в качестве части определения протокола CounterDataSource.

Ниже приведена простая реализация CounterDataSource, где источник данных возвращает постоянное значение 3, каждый раз, как получает запрос. Это осуществляется благодаря тому, что реализуется опциональное требование свойства fixedIncrement:

```
class ThreeSource: NSObject, CounterDataSource {
    let fixedIncrement = 3
}
```

Вы можете использовать экземпляр ThreeSource в качестве источника данных для новых экземпляров Counter:

```
var counter = Counter()
counter.dataSource = ThreeSource()
for _ in 1...4 {
    counter.increment()
    print(counter.count)
}
// 3
// 6
// 9
// 12
```

Draft

Код, приведенный выше, создает новый экземпляр Counter, устанавливает его источник данных как экземпляр ThreeSource и вызывает метод счетчика increment четыре раза. Как и ожидалось, свойство счетчика увеличивается на три каждый раз, когда вызывается increment.

Ниже приведен более сложный источник данных TowardsZeroSource, который заставляет экземпляр Counter считать в сторону увеличения или уменьшения по направлению к нулю от текущего значения count:

```
class TowardsZeroSource: NSObject, CounterDataSource {
    func increment(forCount count: Int) -> Int {
        if count == 0 {
            return 0
        } else if count < 0 {
            return 1
        } else {
            return -1
        }
    }
}
```

Класс TowardsZeroSource реализует опциональный метод increment(forCount:) из протокола CounterDataSource и использует значение аргумента count для определения направления следующего счета. Если count уже ноль, то метод возвращает 0, для отображения того, что дальнейших вычислений не требуется.

Вы можете использовать экземпляр TowardsZeroSource с уже существующим экземпляром Counter для отсчета от -4 и до 0. Как только счетчик достигает 0, вычисления прекращаются:

```
counter.count = -4
counter.dataSource = TowardsZeroSource()
for _ in 1...5 {
    counter.increment()
    print(counter.count)
}
// -3
// -2
// -1
// 0
// 0
```

4.22 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.

БГУИР

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

```
extension RandomNumberGenerator {
    func randomBool() -> Bool {
        return random() > 0.5
    }
}
```

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

```
let generator = LinearCongruentialGenerator()
print("Рандомное число: \(generator.random())")
// Выведет "Рандомное число: 0.37464991998171"
print("Рандомное логическое значение: \(generator.randomBool())")
// Выведет "Рандомное логическое значение: true"
```

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.

4.22.1 Обеспечение реализации по умолчанию (дефолтной реализации)

Вы можете использовать расширение протокола, чтобы обеспечить реализацию по умолчанию для любого метода или требования свойства этого протокола. Если соответствующий тип предоставляет свою собственную реализацию требуемого метода или свойства, то реализация будет использоваться вместо той, которая предоставляется расширением.

Требования протокола с реализацией по умолчанию, предоставляемой расширениями, отличаются от опциональных требований протокола. Хотя соответствующие типы не должны предоставлять свою собственную реализацию, требования с реализацией по умолчанию могут быть вызваны без опциональных последовательностей.

Например, протокол PrettyTextRepresentable, который наследует от протокола TextRepresentable может предоставлять дефолтную реализацию требуемого свойства prettyTextualDescription, просто возвращая результат обращения к свойству textualDescription:

```
extension PrettyTextRepresentable  {
    var prettyTextualDescription: String {
```

```
            return textualDescription
        }
    }
```

4.22.2 Добавление ограничений к расширениям протоколов

Когда вы определяете расширение протокола, вы можете указать ограничения для принимающих типов, которые они должны удовлетворить до того, как будут доступны методы и свойства расширения. Вы записываете эти ограничения сразу после имени протокола, при помощи оговорки where. Более подробно об оговорках where, читайте в разделе "Оговорка Where".

Например, вы можете определить расширение протокола Collection, которое применимо ко всем коллекциям, чьи элементы соответствуют протоколу Equatable. Ограничивая элементы Collection протоколом Equatable, частью стандартной библиотеки, вы можете использовать операторы == и != для проверки равенства и неравенства между двумя элементами.

```
extension Collection where Element: Equatable {
    func allEqual() -> Bool {
        for element in self {
            if element != self.first {
                return false
            }
        }
        return true
    }
}
```

Метод allEqual() возвращает true, только если все элементы в Collection равны.

Рассмотрим два целочисленных массива: один, где все элементы одинаковы, и другой, где элементы различны:

```
let equalNumbers = [100, 100, 100, 100, 100]
let differentNumbers = [100, 100, 200, 100, 200]
```

Поскольку массивы соответствуют Collection и целые числа соответствуют Equatable, equalNumbers и differentNumbers могут использовать метод allEqual():

```
print(equalNumbers.allEqual())
// Prints "true"
print(differentNumbers.allEqual())
// Prints "false"
```

БГУИР

Если подписанный тип удовлетворяет требованиям нескольких ограничивающих расширений, которые предоставляют реализации для одного и того же метода или свойства, то Swift будет использовать самое строгое ограничение.

БГУИР

Вопросы к лабораторной работе № 4

1. Что такое наследование?
2. Что такое переопределение? Что можно переопределять?
3. Что такое протоколы?
4. Для чего служат протоколы?
5. Что такое расширение протоколов?
6. Какой синтаксис у протоколов?
7. Что из себя представляет делигирование?

БГУИР

Задание к лабораторной работе № 4

Задание к лабораторной работе:

согласно варианта задания написать программу (Вариант = номер в списке подгруппы % 15 + 1);

разбить функционал приложения на несколько пакетов придерживаясь логики;

Каждый класс должен реализовывать протокол;

Работа должна происходить не с экземплярами класса, а с экземплярами протокола;

- сделать валидацию всех вводимых значений.

1 Класс рациональных дробей. класс, представляющий рациональную дробь (num – числитель, den – знаменатель). Класс содержит методы умножения и деления (дроби на дробь и дроби на целое число). Создать протокол, который служит для создания случайной дроби из заданного диапазона целых. Добавить две различные реализации протокола.

2 Создать класс, который проверяет число и определяет, что это – простое число, число Фибоначчи, комплексное число, целое, вещественное. Каждый метод относится к отдельному протоколу.

3 Создать протокол Человек, (имя, фамилия, пол, год рождения). Создать не менее 10 объектов данного протокола. Вывести количество по мкжчин и женщин, количество человек на заданный год рождения.

4 Создать класс для вычисления корней квадратного уравнения. Предусмотреть все возможные варианты.

5 Класс Файл. Имя файла, размер, дата создания, количество обращений. Создать список объектов. Вывести: список файлов, упорядоченных по алфавиту, список файлов, размер которых превышает заданный, список файлов, число обращений к которым превышает заданное

6 Класс Train. Пункт назначения, номер поезда, время отправления, число мест(общих, плацкарт, купе) создать список объектов, вывести: список поездов следующих до заданного пункта и количество свободных мест по категориям, список поездов, следующих до заданного пункта и отправляющихся после заданного времени. Создать протокол, который реализует метод по движению поезда.

7 Класс Patient. Фамилия, имя, отчество, возраст, адрес, номер медиц. карты, диагноз (возможно наличие не одного диагноза). Вывести список пациентов (ФИО, адрес, возраст) номера медицинских карт которых находятся в заданном диапазоне, список пациентов с заданным диагнозом

8 Класс Product. Наименование, производитель, цена, срок хранения, количество. Создать список объектов. Вывести список товаров, цена которых не превышает заданной, список товаров с просроченным сроком хранения (название, производитель, срок окончания хранения, количество)

9 Протокол Книга. Создать класс Библеотека, который содержит

БГУИР

список объектов «Книга». Выведите информацию о каждой книге в Библиотеке. Выведите все книги определённого автора. Выведите все книги определённого года выпуска.

10 Протокол Двигатель. Реализовать метод, который запускает двигатель автомобиля. Создать класс, который будет иметь в себе двигатель. Добавить несколько различных моделей двигателей. Создать список автомобилей и запустить их двигатели.

11 Протокол Фрукт. Реализовать метод, который возвращает описание фрукта (например, «Яблоко, цвет: красный, вкус: сладкий»). Создать класс Корзинка, которая будет в себе хранить разничные варианты фруктов. Вывести информацию о всех фруктах в корзинке.

12 Протокол Задача. Реализовать метод, который возвращает информацию о задаче (например, «Задача: Подготовить презентацию, выполнено: нет») и метод, который меняет статус задачи. Создать список задач и вывести их информацию и поменять статус.

13 Создать протокол для выполнения математических операций. Реализовать методы (умножение, сложение, вычитание, деление, возведение в степень). Добавить возможность использовать различный типы численных значений.

14 Протокол Отправки уведомлений. Реализовать метод, который оправляет уведомление. Создать класс, который будет иметь в себе список различных способов доставки уведомлений. Добавить несколько различных способов доставки уведомлений.

15 Протокол Создания уникальных чисел. Реализовать метод, который генерирует уникальное число. Создать класс, который будет иметь в себе генератор. Добавить несколько различных алгоритмов создания уникальных чисел. Создать несколько уникальных последовательностей чисел.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Изучить теоретическую часть лабораторной работы.

2. Реализовать индивидуальное задание по вариантам сделать скриншоты работающих программ. Написать комментарии.

3. Написать отчет, содержащий:

1. Титульный лист, на котором указывается:

а) полное наименование министерства образование и название учебного заведения;

б) название дисциплины;

в) номер практического занятия;

г) фамилия преподавателя, ведущего занятие;

д) фамилия, имя и номер группы студента;

е) год выполнения лабораторной работы.

2. Индивидуальное задание с кодом, комментариями и скриншотами

БГУИР

работающих программ.

3. Вывод по проделанной работе.

БГУИР