**Лабораторная работа №6**

**СТРУКТУРЫ, КЛАССЫ, ПЕРЕЧИСЛЕНИЯ**

**Перечисления (enumerations)**

enum используется для создания перечисления. Каждое отдельное значение в перечислении указывается после оператора case. Как классы и другие именованные типы, перечисления могут иметь ассоциированные с ними методы.

**Вложенные перечисления**

Перечисления могут быть частью других перечислений, то есть могут быть определены в области видимости родительских перечислений.

// тип данных "валюта"

enum CurrencyUnit {

// страны, использующие доллар

enum DollarCountries {

case usa

case canada

case australia

}

case rouble(сountries: [String], shortName: String)

case dollar(сountries: [String], shortName: String, national: DollarCountries)

case euro(сountries: [String], shortName: String)

}

Теперь перечисление DollarCountries обладает ограниченной областью видимости и доступно только через родительское перечисление. Тем не менее при необходимости вы можете создать параметр, содержащий значение этого перечисления, и вне перечисления CurrencyUnit

var someVar: CurrencyUnit.DollarCountries

someVar = .australia

**Ассоциированные значения**

С каждым значением в перечислении мы можем ассоциировать какое-либо другое значения, которое может представлять любой тип. При этом ассоциированные значения для разных значений одного и того же перечисления могут представлять различные типы.

enum Smile {

case joy (String)

case laugh (String, Int)

}

Для получения значений мы можем определить константы или переменные, в которые будут передаваться ассоциированные значения.

**Связанные значения членов перечисления**

Как альтернативу ассоциированным параметрам для членов перечислений им можно задать связанные значения некоторого типа данных (например, String, Character или Int). В результате вы получаете член перечисления и постоянно привязанное к нему значение.

Для задания связанных значений членов перечисления необходимо указать тип данных самого перечисления, соответствующий значениям членов, и определить значения для каждого отдельного члена перечисления

enum Smile: String {

case joy = ":)"

case laugh = ":D"

case sorrow = ":("

case surprise = "o\_O"

}

Связанные значения и ассоциированные параметры — не одно и то же. Исходные значения устанавливаются при определении перечисления, причем обязательно для всех его членов и в одинаковом типе данных. Ассоциированные параметры могут быть различны для каждого перечисления и устанавливаются лишь при создании экземпляра этого перечисления.

При создании экземпляра перечисления можно получить доступ к исходному значению члена этого экземпляра перечисления. Для этого используется свойство rawValue. Создадим экземпляр объявленного ранее перечисления Smile и получим исходное значение установленного в этом экземпляре члена

var iAmHappy = Smile.joy

iAmHappy.rawValue //:)

В результате использования свойства rawValue мы получаем исходное значение члена joy типа String.

Свойство rawValue может быть использовано не только для получения связанного значения, но и для создания экземпляра перечисления. Для этого необходимо вызвать инициализатор перечисления и передать ему требуемое исходное значение.

enum Planet: Int {

case mars = 1

case alfa = 3

case betta = 11

case splus = 5

}

var myPlanet = Planet.init(rawValue: 3) //alfa

var anotherPlanet = Planet.init(rawValue: 11) //betta

Инициализатор — это метод init(rawValue:) перечисления Planet. Ему передается указатель на исходное значение, связанное с искомым членом этого перечисления. Не удивляйтесь, что метод init(rawValue:) не описан в пределах перечисления, — он существует там всегда по умолчанию и закреплен в исходном коде языка Swift. Инициализатор init(rawValue:) возвращает опционал, поэтому если вы укажете несуществующее связанное значение, возвратится nil.

**Свойства в перечислениях**

Благодаря перечислениям можно смоделировать ситуацию, в которой существует ограниченное количество исходов. У таких ситуаций помимо возможных результатов (членов перечисления) могут существовать и некоторые свойства. Свойство в перечислении — это хранилище, аналогичное переменной или константе, объявленное в пределах перечисления и доступное через доступ к его экземпляру. В Swift существует определенное ограничение для свойств в перечислениях: в качестве их значений не могут выступать фиксированные значения-литералы, а лишь замыкания. Такие свойства называются вычисляемыми. При каждом обращении к ним происходит вычисление присвоенного замыкания с возвращением получившегося значения. Для вычисляемого свойства после имени через двоеточие указывается тип возвращаемого значения и далее без оператора присваивания в фигурных скобках — тело замыкающего выражения, генерирующего возвращаемое значение.

enum Smile: String {

case joy = ":)"

case laugh = ":D"

case sorrow = ":("

case surprise = "o\_O"

var description: String {return self.rawValue} //:(

}

var mySmile: Smile = .sorrow

mySmile.description //:(

Вычисляемое свойство должно быть объявлено как переменная (var). В противном случае (если используете оператор let) вы получите сообщение об ошибке. С помощью оператора self вы получаете доступ к текущему члену перечисления.

**Методы в перечислениях**

Перечисления могут группировать в себе не только свойства, члены и другие перечисления, но и методы. Ранее мы говорили об инициализаторах init(), которые являются встроенными в перечисления методами . Методы — это функции внутри перечислений, поэтому их синтаксис и возможности идентичны синтаксису и возможностям изученных ранее функций.

enum Smile: String {

case joy = ":)"

case laugh = ":D"

case sorrow = ":("

case surprise = "o\_O"

func description(){

print("Перечисление содержит список используемых смайликов: их названия и графические обозначения")

}

}

var mySmile = Smile.joy

mySmile.description()

В этом перечислении объявлен метод description(). После создания экземпляра метода и помещения его в переменную метод description() можно вызвать.

**Оператор self**

Методы могут не только совершать какие-либо не относящиеся к перечислению вещи, но и обрабатывать его члены и связанные значения. Для доступа к значению экземпляра внутри самого перечисления используется оператор self, который возвращает указатель на данный экземпляр перечисления.

enum Smile: String {

case joy = ":)"

case laugh = ":D"

case sorrow = ":("

case surprise = "o\_O"

func description(){

print("Перечисление содержит список используемых смайлов: их названия и графические обозначения")

}

func descriptionValue() -> Smile{

return self

}

func descriptionRawValue() -> String{

return self.rawValue

}

}

var mySmile = Smile.joy

mySmile.descriptionValue()

mySmile.descriptionRawValue()

При вызове метода descriptionValue() происходит возврат self, то есть самого экземпляра. Именно поэтому тип возвращаемого значения данного метода — Smile, он соответствует типу экземпляра перечисления. Метод descriptionRawValue() возвращает связанное значение члена данного экземпляра также с использованием оператора self. При необходимости вы даже можете выполнить анализ перечисления внутри самого перечисления с помощью конструкции switch self {}, где значениями являются члены перечисления. Оператор self можно использовать не только для перечислений, но и для структур и классов.

**Рекурсивные перечисления**

Возьмем, к примеру, простейшие арифметические операции: сложение, вычитание, умножение и деление. Все они заранее известны, поэтому могут быть помещены в перечисление в качестве его членов.

Для простоты рассмотрим только две операции: сложение и вычитание.

enum ArithmeticExpression{

// операция сложения

case addition(Int, Int)

// операция вычитания

case substraction(Int, Int)

}

var expr = ArithmeticExpression.addition(10, 14)

Данное перечисление имеет два члена, как и положено в арифметических выражениях: арифметические операции в Swift — это бинарные операторы, то есть операторы, проводящие операцию с двумя операндами. В связи с этим каждая из операций может принимать два ассоциированных параметра. В результате в переменной expr хранится член перечисления ArithmeticExpression, определяющий арифметическую операцию сложения. Объявленное перечисление не несет какой-либо функциональной нагрузки в вашем приложении. Но вы можете создать в его пределах метод, который определяет наименование члена и возвращает результат данной операции

enum ArithmeticExpression{

case addition(Int, Int)

case substraction(Int, Int)

// метод подсчета

func evaluate() -> Int {

switch self{

case .addition(let left, let right):

return left + right

case .substraction(let left, let right):

return left - right

}

}

}

var expr = ArithmeticExpression.addition(10, 14)

expr.evaluate() //24

При вызове метода evaluate() происходит поиск определенного в данном экземпляре члена перечисления. Для этого используются операторы switch и self. Далее, после того как член определен, путем связывания значений возвращается результат данной арифметической операции

Данный способ работает, но имеет серьезное ограничение: он способен моделировать только одноуровневые арифметические выражения: 1 + 5, 6 + 19 и т. д . В ситуации, когда выражение имеет вложенные выражения: 1 + (5 – 7), 6 – 5 + 4 и т.д., нам придется вычислять каждое отдельное действие с использованием собственного экземпляра. Для решения этой проблемы необходимо доработать перечисление ArithmeticExpression таким образом, чтобы оно давало возможность складывать не только значения типа Int, но и значения типа ArithmeticExpression. Получается, что перечисление, описывающее выражение, должно давать возможность выполнять операции само с собой. Данный механизм реализуется в рекурсивном перечислении. Для того чтобы разрешить членам перечисления обращаться к этому перечислению, используется ключевое слово indirect, которое ставится:

* либо перед оператором enum — в этом случае каждый член перечисления может обратиться к данному перечислению;
* либо перед оператором case того члена, в котором необходимо обратиться к перечислению.

Если в качестве ассоциированных параметров перечисления указывать значения типа самого перечисления ArithmeticExpression, то возникает вопрос: а где же хранить числа, над которыми совершаются операции? Такие числа также необходимо хранить в самом перечислении, в его отдельном члене. Рассмотрим пример. В данном примере вычисляется значение выражения 20 + 10 – 34.

enum ArithmeticExpression {

// хранилище для операнда

case number(Int)

indirect case addition(ArithmeticExpression, ArithmeticExpression)

indirect case subtraction(ArithmeticExpression, ArithmeticExpression)

func evaluate(expression: ArithmeticExpression? = nil ) -> Int{

let expression = (expression == nil ? self : expression)

switch expression! {

case .number( let value ):

return value

case .addition( let valueLeft, let valueRight ):

return self.evaluate( expression: valueLeft ) + self.evaluate( expression: valueRight )

case .subtraction( let valueLeft, let valueRight ):

return self.evaluate( expression: valueLeft ) - self.evaluate( expression: valueRight )

}

}

}

var expr = ArithmeticExpression.addition( .number(20), .subtraction( .number(10), .number(34) ) )

expr.evaluate() // -4

Здесь у перечисления появился новый член number. Он определяет целое число — операнд для проведения операции. Для членов арифметических операций использовано ключевое слово indirect, позволяющее передать значение типа ArithmeticExpression в качестве ассоциированного параметра. Метод evaluate(expression:) принимает на входе опциональное значение типа ArithmeticExpression?. Опционал в данном случае позволяет вызвать метод, не передавая ему экземпляр, из которого этот метод был вызван. В противном случае последняя строка листинга выглядела бы следующим образом:

expr.evaluate(expression: expr)

Согласитесь, что существующий вариант значительно удобнее. Оператор switch, используя принудительное извлечение, определяет, какой член перечисления передан, и возвращает соответствующее значение. В результате данное перечисление позволяет смоделировать любую операцию, в которой присутствуют операторы сложения и вычитания. Перечисления в Swift мощнее, чем аналогичные механизмы в других языках программирования. Вы можете создавать свойства и методы, применять к ним расширения и протоколы, а также делать многое другое.

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

**Задание 6.1**

1. *Создайте перечисление с месяцами. Напишите в перечислении функцию, которая возвращает по месяцу пору года. Члены перечисления должны иметь связанные значения.*
2. *Используйте перечисление для установки месяца. Определите пору года.*
3. *Допишите перечисление ArithmeticExpression таким образом, чтобы оно могло реализовать любое выражение с использованием операций сложения, вычитания, умножения, деления и возведения в степень.*

**Структуры**

Все фундаментальные типы данных — коллекции — это структуры. Структуры в некоторой степени похожи на перечисления и во многом сходны с классами.

Используйте struct, чтобы создать структуру.

Свойство — это характеристика сущности, которую описывает структура. Оно может быть переменной или константой и объявляется в теле структуры.

struct PlayerInChess {

var name: String

var wins: Int

}

Структуры, так же, как и перечисления, имеют встроенный инициализатор init, который не требуется объявлять. Данный инициализатор принимает на входе значения всех свойств структуры, производит их инициализацию и создает экземпляр структуры

var oleg = PlayerInChess(name: "Олег", wins: 32)

Для свойств можно задавать значение по умолчанию. При этом Swift автоматически создает новый инициализатор, который позволяет создавать экземпляр без указания значений свойств.

Значения по умолчанию указываются вместе с объявлением свойств точно так же, как вы указываете значение любой переменной или константы. При этом если вы решили дать значение по умолчанию хотя бы одному свойству, то должны указывать его и для всех остальных свойств. Swift позволяет определять значения по умолчанию лишь для некоторых свойств только в случае создания собственного иницализатора.

struct PlayerInChess {

var name = ""

var wins: Int = 0

}

var oleg = PlayerInChess(name: "Олег", wins: 32)

var maks = PlayerInChess()

Если вас не устраивают инициализаторы, которые создаются для структур автоматически, вы имеете возможность разработать собственные. Вам необходимо постоянно придерживаться правила: «все свойства структуры должны иметь значения». Вы можете создать инициализатор, который принимает в качестве входного параметра значения не для всех свойств, тогда остальным свойствам должны быть назначены значения либо внутри данного инициализатора, либо через значения по умолчанию. При этом одна структура может содержать произвольное количество инициализаторов, каждый из которых должен иметь уникальный набор входных параметров.

struct PlayerInChess {

var name: String

var wins: Int = 0

init(name: String){

self.name = name

}

}

var oleg = PlayerInChess(name: "Олег")

При создании нового экземпляра вам будет доступен только разработанный вами инициализатор. В него передается значение параметра name. Доступ к свойствам экземпляра для их изменения осуществляется с помощью оператора self. Свойство wins имеет значение по умолчанию, оно присваивается данному свойству автоматически.

Структуры — это типы-значения. При передаче экземпляра структуры от одного параметра в другой происходит его копирование. В следующем примере создаются два независимых экземпляра одной и той же структуры.

var olegMuhin = PlayerInChess(name: "Олег")

var olegLapin = olegMuhin

**Методы в структурах**

Они, как и обычные функции, могут принимать входные параметры. Для доступа к собственным свойствам структуры используется оператор self.

struct PlayerInChess {

var name: String

var wins: Int

func description(){

print("Игрок \(self.name) имеет \(self.wins) побед")

}

}

var oleg = PlayerInChess(name: "Олег", wins: 15)

oleg.description()

По умолчанию методы структур, кроме инициализаторов, не могут изменять значения свойств, объявленные в тех же самых структурах. Для того чтобы обойти данное ограничение, перед именем объявляемого метода необходимо указать модификатор mutating.

struct PlayerInChess {

var name: String

var wins: Int

// метод, изменяющий значение свойства wins

mutating func addWins(\_ countOfWins: Int ){

self.wins += countOfWins

}

}

var oleg = PlayerInChess(name: "Олег", wins: 15)

oleg.wins

oleg.addWins(3)

oleg.wins

**Классы**

Объявление свойства класса делается так же, как объявление обычной переменной или константы — с тем лишь исключением, что оно теперь находится в контексте класса. Аналогично, методы и функции задаются внутри класса:

class Shape {

        var numberOfSides = 0

        func simpleDescription() -> String {

                return "A shape with \(numberOfSides) sides."

        }

}

Создается объект на основе класса, используя скобки после имени класса. Используйте синтаксис с точкой для доступа к свойствам и методам объекта:

var shape = Shape()

shape.numberOfSides = 7

var shapeDescription = shape.simpleDescription()

При создании экземпляра класса каждое свойство должно иметь определенное значение, а любое свойство, для которого не указано значение по умолчанию, должно получать значение с помощью разработанного вами инициализатора.

// типы фигур

enum ChessmanType {

case king

case castle

case bishop

case pawn

case knight

case queen

}

// цвета фигур

enum ChessmanColor {

case black

case white

}

class Chessman {

let type: ChessmanType

let color: ChessmanColor

var coordinates: (String, Int)? = nil

let figureSymbol: Character

init(type: ChessmanType, color: ChessmanColor, figure: Character){

self.type = type

self.color = color

self.figureSymbol = figure

}

// метод установки координат

func setCoordinates(char c:String, num n: Int){

self.coordinates = (c, n)

}

// метод, убивающий фигуру

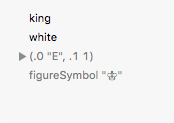
func kill(){

self.coordinates = nil

}

}

var kingWhite = Chessman(type: .king, color: .white, figure: "♔")



kingWhite.setCoordinates(char: "E", num: 1)

class Chessman {

enum ChessmanType {

case king

case castle

case bishop

case pawn

case knight

case queen

}

enum ChessmanColor {

case black

case white

}

let type: ChessmanType

let color: ChessmanColor

var coordinates: (String, Int)? = nil

let figureSymbol: Character

init(type: ChessmanType, color: ChessmanColor, figure: Character){

self.type = type

self.color = color

self.figureSymbol = figure

}

init(type: ChessmanType, color: ChessmanColor, figure: Character, coordinates: (String, Int)){

self.type = type

self.color = color

self.figureSymbol = figure

self.setCoordinates(char: coordinates.0, num: coordinates.1)

}

func setCoordinates(char c:String, num n: Int){

self.coordinates = (c, n)

}

func kill(){

self.coordinates = nil

}

}

В некоторых ситуациях может возникнуть необходимость использовать вложенные типы вне определяющего их контекста. Для этого необходимо указать имя родительского типа, после которого должно следовать имя требуемого типа данных

**Типы свойств**

По типу хранимого значения можно выделить два основных вида свойств:

* хранимые свойства могут использоваться в структурах и классах;
* вычисляемые свойства могут использоваться в перечислениях, структурах и классах.

Хранимое свойство — это константа или переменная, объявленная в объектном типе и хранящая определенное значение. Хранимое свойство обладает следующими возможностями:

* получает значение по умолчанию в случае, если при создании экземпляра ему не передается никакого значения;
* получает значение в инициализаторе, передаваемое при создании экземпляра в качестве входного аргумента. Данное значение называется исходным;
* меняет значение в процессе использования экземпляра.

**Ленивые хранимые свойства**

Хранимые свойства могут быть «ленивыми». Значение, которое должно храниться в ленивом свойстве, не создается до момента первого обращения к нему .

СИНТАКСИС

lazy var имяСвойства1

lazy let имяСвойства2

class AboutMan{

let firstName = "Егор"

var secondName = "Петров"

lazy var wholeName: String = self.generateWholeName()

func generateWholeName() -> String{

return self.firstName + " " + self.secondName

}

}

var Me = AboutMan()

Me.wholeName

В свойстве wholeName должно храниться его полное имя, но при создании экземпляра значения этого свойства не существует. При этом оно не равно nil, оно просто не сгенерировано и не записано. Это связано с тем, что свойство является ленивым. Как только происходит обращение к данному свойству, его значение формируется.

Ленивые свойства являются lazy-by-need, то есть вычисляются однажды и больше не меняют свое значени

var Me2 = AboutMan()

Me2.wholeName // Егор Петров

Me2.secondName = "Иванов"

Me2.wholeName // Егор Петров

**Вычисляемые свойства**

С вычисляемыми свойствами мы уже встречались у перечислений, для экземпляров которых это единственный доступный вид свойств. Вычисляемые свойства фактически не хранят значение, а вычисляют его с помощью замыкающего выражения.

СИНТАКСИС

var имяСвойства: ТипДанных { тело\_замыкающего\_выражения }

Вычисляемые свойства могут храниться исключительно в переменных(var). После указания имени объявляемого свойства и типа возвращаемого замыкающим выражением значения без оператора присваивания указывается замыкание, в результате которого должно быть сгенерировано возвращаемое свойством значение. Для того чтобы свойство возвращало некоторое значение, в теле замыкания должен присутствовать оператор return.

class AboutMan{

let firstName = "Егор"

var secondName = "Петров"

var wholeName: String {return self.firstName + " " + self.secondName}

}

var Me = AboutMan()

Me.wholeName //Егор Петров

Значение свойства wholeName идентично соответствующему значению из предыдущего примера, за тем лишь исключением, что оно будет вычислять результат каждый раз при доступе к нему.

**Контроль получения и установки значений**

*Геттер и сеттер вычисляемого свойства*

Для любого вычисляемого свойства существует возможность реализовать две специальные функции:

* Геттер (get) выполняет некоторый код при запросе значения вычисляемого свойства.
* Сеттер (set) выполняет некоторый код при попытке установки значения вычисляемого свойства.

Во всех объявленных ранее вычисляемых свойствах был реализован только геттер, поэтому они являлись свойствами «только для чтения», то есть попытка изменения вызвала бы ошибку. Таким образом, можно утверждать, что геттер является обязательным для существования вычисляемого свойства, а реализацию сеттера можно опустить.

СИНТАКСИС

var имяСвойства1: ТипЗначения {

get {

тело\_геттера

}

set (передаваемый\_параметр) {

тело\_сеттера

}

}

struct Circle{

var coordinates: (x: Int, y: Int)

var radius: Float

var perimeter: Float {

get{

return 2.0\*3.14\*self.radius

}

set(newPerimeter){

self.radius = newPerimeter / (2\*3.14)

}

}

}

var myNewCircle = Circle(coordinates: (0,0), radius: 10)

myNewCircle.perimeter // выводит 62.8

myNewCircle.perimeter = 31.4

myNewCircle.radius // выводит 5

Сеттер также позволяет использовать сокращенный синтаксис записи, в котором не указывается имя входного параметра. При этом внутри сеттера для доступа к устанавливаемому значению необходимо задействовать автоматически объявляемый параметр с именем newValue.

struct Circle{

var coordinates: (x: Int, y: Int)

var radius: Float

var perimeter: Float {

get{

return 2.0\*3.14\*self.radius

}

set{

self.radius = newValue / (2\*3.14)

}

}

}

**Наблюдатели хранимых свойств**

Геттер и сеттер позволяют выполнять код при установке и чтении значения вычисляемого свойства. Другими словами, у вас имеются механизмы контроля процесса изменения и чтения значений. Наделив такими полезными механизмами вычисляемые свойства, разработчики Swift не могли обойти стороной и хранимые свойства. Специально для них были реализованы наблюдатели (observers). Наблюдатели — это специальные функции, которые вызываются либо непосредственно перед, либо сразу после установки нового значения хранимого свойства. Выделяют два вида наблюдателей:

* Наблюдатель willSet вызывается перед установкой нового значения.
* Наблюдатель didSet вызывается после установки нового значения.

СИНТАКСИС

var имяСвойства1: ТипЗначения {

willSet (параметр){

тело\_геттера

}

didSet (параметр){

тело\_сеттера

}

}

При объявлении наблюдателей можно использовать сокращенный синтаксис, в котором не требуется указывать входные аргументы (точно так же, как сокращенный синтаксис сеттера). При этом новое значение в willSet присваивается параметру newValue, а старое в didSet — параметру oldValue.

struct Circle{

var coordinates: (x: Int, y: Int)

var radius: Float {

willSet (newValueOfRadius) {

print("Вместо значения \(self.radius) устанавливается значение \(newValueOfRadius)")

}

didSet (oldValueOfRadius) {

print("Вместо значения \(oldValueOfRadius) установлено значение \(self.radius)")

}

}

var perimeter: Float {

get{

return 2.0\*3.14\*self.radius

}

set{

self.radius = newValue / (2\*3.14)

}

}

}

var myNewCircle = Circle(coordinates: (0,0), radius: 10)

myNewCircle.perimeter // выведет 62.8

myNewCircle.perimeter = 31.4

//Вместо значения 10.0 устанавливается значение 5.0

//Вместо значения 10.0 установлено значение 5.0

myNewCircle.radius // выведет 5

**Свойства типа**

Дополнительно к свойствам экземпляров вы можете объявлять свойства, относящиеся к типу данных. Значения этих свойств едины для всех экземпляров типа. Свойства типа данных очень полезны в том случае, когда существуют значения, которые являются универсальными для всего типа целиком. Они могут быть как хранимыми, так и вычисляемыми. При этом если значение хранимого свойства типа является переменной и изменяется в одном экземпляре, то измененное значение становится доступно во всех других экземплярах типа.

ВНИМАНИЕ Для хранимых свойств типа в обязательном порядке должны быть указаны значения по умолчанию. Это связано с тем, что сам по себе тип не имеет инициализатора, который бы мог сработать еще во время определения типа и установить требуемые значения для свойств. Хранимые свойства типа всегда являются ленивыми, при этом они не нуждаются в использовании ключевого слова lazy.

Свойства типа могут быть созданы для перечислений, структур и классов.

struct SomeStructure {

static var storedTypeProperty = "Some value"

static var computedTypeProperty: Int {

return 1

}

}

enum SomeEnumiration{

static var storedTypeProperty = "Some value"

static var computedTypeProperty: Int {

return 2

}

}

class SomeClass{

static var storedTypeProperty = "Some value."

static var computedTypeProperty: Int {

return 3

}

class var overrideableComputedTypeProperty: Int {

return 4

}

}

struct AudioChannel {

static var maxVolume = 5

var volume: Int {

didSet {

if volume > AudioChannel.maxVolume {

volume = AudioChannel.maxVolume

}

}

}

}

var LeftChannel = AudioChannel(volume: 2)

var RightChannel = AudioChannel(volume: 3)

RightChannel.volume = 6

RightChannel.volume // выводит 5

AudioChannel.maxVolume // выводит 5

AudioChannel.maxVolume = 10

AudioChannel.maxVolume // выводит 10

RightChannel.volume = 8

RightChannel.volume // выводит 8

**Сабскрипты**

Мы встречались с понятием сабскрипта, это указываемый для доступа к определенному значению индекс. Однако сабскрипты позволяют также упростить работу со структурами и классами.

С помощью сабскриптов структуры и классы можно превратить в некое подобие коллекций, то есть мы сможем обращаться к свойствам экземпляра с использованием ключей.

Возьмем класс Chessboard, моделирующий сущность «шахматная доска». Экземпляр класса Chessboard хранится в переменной desk:

var desk: Chessboard

В одном из свойств данного экземпляра содержится информация о том, какая клетка поля какой шахматной фигурой занята. Для доступа к информации относительно определенной клетки мы можем разработать специальный метод, которому в качестве входных параметров будут передаваться координаты:

desk.getCellInfo("A", 5)

С помощью сабскриптов можно организовать доступ к ячейкам клетки, передавая координаты, подобно ключу массива, непосредственно экземпляру класса:

desk["A", 5]

СИНТАКСИС

subscript(входные\_параметры) —> тип\_возвращаемого\_значения {

get{

// тело геттера

}

set(новое\_значение){

// тело сеттера

}

}

Сабскрипты объявляются в теле класса или структуры с помощью ключевого слова subscript. Далее указываются входные аргументы (в точности также, как у методов) и тип значения. Входные аргументы —это значения, которые передаются в виде ключей. Тип значения указывает на тип данных устанавливаемого (в случае сеттера) или возвращаемого (в случае геттера) значения. Тело сабскрипта заключается в фигурные скобки и состоит из геттера и сеттера по аналогии с вычисляемыми значениями. Геттер выполняет код при запросе значения с использованием сабскрипта, сеттер—при попытке установитьз начение.

Сеттер также дает возможность дополнительно указать имя параметра, которому будет присвоено новое значение. Если данный параметр не будет указан, то новое значение автоматически инициализируется локальной переменной newValue. При этом тип данных параметра будет соответствовать типу возвращаемого значения. *Сеттер является необязательным*, и в случае его отсутствия может быть использован сокращенный синтаксис:

subscript(входные\_параметры) —> возвращаемое\_значение

{ // тело геттера

}

Сабскрипты поддерживают *перегрузку*, то есть в пределах одного объектного типа может быть определено произвольное количество сабскриптов, различающихся лишь набором входных аргументов.

При разработке класса реализуем метод, устанавливающий переданную ему фигуру на игровое поле. При этом стоит помнить о двух моментах:

* фигура, возможно, уже находилась на поле, а значит, ее требуется удалить со старой позиции;
* фигура имеет свойство сoordinates, которое также необходимо изменять.

class Chessman {

enum ChessmanType {

case king

case castle

case bishop

case pawn

case knight

case queen

}

enum ChessmanColor {

case black

case white

}

let type: ChessmanType

let color: ChessmanColor

var coordinates: (String, Int)? = nil

let figureSymbol: Character

init(type: ChessmanType, color: ChessmanColor, figure: Character){

self.type = type

self.color = color

self.figureSymbol = figure

}

init(type: ChessmanType, color: ChessmanColor, figure: Character, coordinates: (String, Int)){

self.type = type

self.color = color

self.figureSymbol = figure

self.setCoordinates(char: coordinates.0, num: coordinates.1)

}

func setCoordinates(char c:String, num n: Int){

self.coordinates = (c, n)

}

func kill(){

self.coordinates = nil

}

}

class gameDesk {

var desk: [Int:[String:Chessman]] = [:]

init(){

for i in 1...8 {

desk[i] = [:]

}

}

func setChessman(chess: Chessman, coordinates: (String, Int)){

if let oldCoordinates = chess.coordinates {

desk[oldCoordinates.1]![oldCoordinates.0] = nil

}

self.desk[coordinates.1]![coordinates.0] = chess

chess.setCoordinates(char: coordinates.0, num: coordinates.1)

}

}

var game = gameDesk()

var QueenBlack = Chessman(type: .queen, color: .black, figure: "♛", coordinates: ("A", 6))

game.setChessman(chess: QueenBlack, coordinates: ("B",2))

game.setChessman(chess: QueenBlack, coordinates: ("A",3))

Класс gameDesk описывает игровое поле. Его основным и единственным свойством является коллекция клеток, на которых могут располагаться шахматные фигуры (экземпляры класса Chessman).

При создании экземпляра свойству desk устанавливается значение по умолчанию «пустой словарь». Во время работы инициализатора в данный словарь записываются значения, соответствующие номерам строк на шахматной доске. Это делается для того, чтобы обеспечить безошибочную работу при установке фигуры на шахматную клетку. В противном случае при установке фигуры нам пришлось бы сначала узнать состояние линии (существует ли она в словаре), а уже потом записывать фигуру на определенные координаты. Метод setChessman(\_:coordinates:) не просто устанавливает ссылку на фигуру в свойство desk, но и убирает старую ссылку. Это обеспечивается тем, что актуальные координаты всегда хранятся в свойстве coordinates экземпляра класса Chessman. В написанном классе отсутствует возможность запроса информации о произвольной ячейке. Реализуем ее с использованием сабскрипта. В сабскрипт будут передаваться координаты необходимой ячейки в виде двух отдельных входных аргументов. Если по указанным координатам существует фигура, то она возвращается, в противном случае возвращается nil.

class gameDesk {

var desk: [Int:[String:Chessman]] = [:]

init(){

for i in 1...8 {

desk[i] = [:]

}

}

subscript(alpha: String, number: Int) -> Chessman? {

get{

if let chessman = self.desk[number]![alpha] {

return chessman

}

return nil

}

}

func setChessman(chess: Chessman, coordinates: (String, Int)){

if let oldCoordinates = chess.coordinates {

desk[oldCoordinates.1]![oldCoordinates.0] = nil

}

self.desk[coordinates.1]![coordinates.0] = chess

chess.setCoordinates(char: coordinates.0, num: coordinates.1)

}

}

var game = gameDesk()

var QueenBlack = Chessman(type: .queen, color: .black, figure: "♛", coordinates: ("A", 6))

game.setChessman(chess: QueenBlack, coordinates: ("A",3))

game["A",3]?.coordinates //Optional(("A", 3))

Реализованный сабскрипт имеет только геттер, причем в данном случае можно было использовать краткий синтаксис записи (без ключевого слова get). Так как сабскрипт возвращает опционал, перед доступом к свойству coordinates возвращенной шахматной фигуры необходимо выполнить извлечение опционального значения. Теперь мы имеем возможность установки фигур на шахматную доску с помощью метода setChessman(\_:coordinates:) и получения информации об отдельной ячейке с помощью сабскрипта . Мы можем расширить функционал сабскрипта, добавив в него сеттер, позволяющий устанавливать фигуру на новые координаты

class gameDesk {

var desk: [Int:[String:Chessman]] = [:]

init(){

for i in 1...8 {

desk[i] = [:]

}

}

subscript(alpha: String, number: Int) -> Chessman? {

get{

if let chessman = self.desk[number]![alpha] {

return chessman

}

return nil

}

set{

self.setChessman(chess: newValue!, coordinates: (alpha, number))

}

}

func setChessman(chess: Chessman, coordinates: (String, Int)){

if let oldCoordinates = chess.coordinates {

desk[oldCoordinates.1]![oldCoordinates.0] = nil

}

self.desk[coordinates.1]![coordinates.0] = chess

chess.setCoordinates(char: coordinates.0, num: coordinates.1)

}

}

var game = gameDesk()

var QueenBlack = Chessman(type: .queen, color: .black, figure: "♛", coordinates: ("A", 6))

game.setChessman(chess: QueenBlack, coordinates: ("A",3))

game["C",5] = QueenBlack

QueenBlack.coordinates //Optional(("C", 5))

Запомните, что сабскрипты не могут использоваться как хранилища,то есть через них мы организуем только доступ к хранилищам значений.

**Наследование**

СИНТАКСИС

class SomeSubClass: SomeSuperClass {

// тело подкласса

}

Доступ к наследуемым элементам родительского класса в производном классе реализуется так же, как к собственным элементам данного производного класса, то есть с использованием ключевого слова self.

Для класса безразлично, с какими характеристиками он взаимодействует, собственными или наследуемыми. Данное утверждение справедливо до тех пор, пока не меняется реализация наследуемых характеристик.

**Переопределение наследуемых элементов**

Субкласс может создавать собственные реализации свойств, методов и сабскриптов, наследуемых от суперкласса. Такие реализации называются переопределением. Для переопределения параметров суперкласса в Swift необходимо указать ключевое слово override перед определением элемента.

class Dog{

func bark(){

print (" ")

}

}

class NoisyDog: Dog{

override func bark(){

print ("woof")

print ("woof")

print ("woof")

}

}

var badDog = NoisyDog()

badDog.bark() // выводит woof woof woof

Несмотря на то что переопределение изменяет реализацию свойств, методов и сабскриптов, Swift позволяет осуществлять доступ внутри производного класса к переопределенным элементам суперкласса. Для этого в качестве префикса имени элемента вместо self используется ключевое слово super.

class NoisyDog: Dog{

override func bark(){

for \_ in 1...3 {

super.bark()

}

}

}

var badDog = NoisyDog()

badDog.bark() // выводит woof woof woof

**Переопределение инициализаторов**

Инициализаторы являются такими же наследуемыми элементами, как и методы. Если в подклассе набор свойств, требующих установки значений, не отличается, то наследуемый инициализатор может быть использован для создания экземпляра подкласса. Тем не менее вы можете создать собственную реализацию наследуемого инициализатора. Запомните, что если вы определяете инициализатор с уникальным для суперкласса и подкласса набором входных аргументов, то вы не переопределяете инициализатор, а объявляете новый. Если *подкласс имеет хотя бы один собственный инициализатор, то инициализаторы родительского класса не наследуются*.

class Quadruped{

var name:String;

var type:String;

init(){

self.name="noname"

self.type=" "

}

}

class Dog: Quadruped {

override init(){

super.init()

self.type = "dog"

}

func bark(){

print("woof")

}

func printName(){

print(self.name)

}

}

var myDog = Dog();

myDog.printName();

Прежде чем получать доступ к наследуемым свойствам в переопределенном инициализаторе, необходимо вызвать инициализатор родителя. Он выполняет инициализацию всех наследуемых свойств. Если в подклассе есть собственные свойства, которых нет в суперклассе, то их значения в инициализаторе необходимо указать до вызова инициализатора родительского класса.

**Переопределение наследуемых свойств**

Как отмечалось ранее, вы можете переопределять любые наследуемые элементы. Наследуемые свойства иногда ограничивают функциональные возможности субкласса. В таком случае можно переписать геттер или сеттер данного свойства или при необходимости добавить наблюдатель.

С помощью механизма переопределения вы можете расширить *наследуемое «только для чтения» свойство до «чтение–запись»*, реализовав в нем и геттер и сеттер. Но обратное невозможно: если у наследуемого свойства реализованы и геттер и сеттер, вы не сможете сделать из него свойство «только для чтения» .

ВНИМАНИЕ *Хранимые свойства переопределять нельзя*, так как вызываемый или наследуемый инициализатор родительского класса попытается установить их значения, но не найдет их.

Субкласс не знает деталей реализации наследуемого свойства в суперклассе, он знает лишь имя и тип наследуемого свойства. Поэтому необходимо всегда указывать и имя и тип переопределяемого свойства.

**Превентивный модификатор final**

Swift позволяет защитить реализацию класса целиком или его отдельных элементов. Для этого необходимо использовать превентивный модификатор final, который указывается перед объявлением класса или его отдельных элементов:

* final class для классов;
* final var для свойств;
* final func для методов;
* final subscript для сабскриптов.

При защите реализации класса его наследование в другие классы становится невозможным. Для элементов класса их наследование происходит, но переопределение становится недоступным.

**Подмена экземпляров классов**

Наследование, помимо всех перечисленных возможностей, позволяет заменять требуемые экземпляры определенного класса экземплярами одного из подклассов.

**Приведение типов**

Для анализа классов в единой иерархии существует специальный механизм, называемый приведением типов.

Путем приведения типов вы можете выполнить следующие операции:

* проверить тип конкретного экземпляра класса на соответствие некоторому типу или протоколу;
* преобразовать тип конкретного экземпляра в другой тип той же иерархии классов.

*Проверка типа*

Проверка типа экземпляра класса производится с помощью оператора is. Данный оператор возвращает true в случае, когда тип проверяемого экземпляра является указанным после оператора классом или наследует его

for item in animalsArray {

if item is Dog {

print("Yes")

}

}

*Преобразование типа*

Как неоднократно отмечалось ранее, объявленный и наполненный массив animalsArray имеет элементы разных типов данных из одной иерархической структуры. Несмотря на это, при получении очередного элемента вы будете работать исключительно с использованием методов класса, указанного в типе массива (в данном случае Quadruped). То есть, получив элемент типа Dog, вы не увидите определенный в нем метод bark(), поскольку Swift подразумевает, что вы работаете именно с экземпляром типа Quadruped. Для того чтобы преобразовать тип и сообщить Swift, что данный элемент является экземпляром определенного типа, используется оператор as, точнее, две его вариации: as? и as!.

Данный оператор ставится после имени экземпляра, а после него указывает имя класса, в который преобразуется экземпляр. Между обеими формами оператора существует разница:

* as? ИмяКласса - возвращает либо экземпляр типа ИмяКласса? (опционал), либо nil в случае неудачного преобразования;
* as! ИмяКласса - производит принудительное извлечение значения и возвращает экземпляр типа ИмяКласса или вызывает ошибку в случае неудачи.

ВНИМАНИЕ *Тип данных может быть преобразован только в пределах собственной иерархии классов*.

for item in animalsArray {

if var animal = item as? NoisyDog {

animal.bark()

}else if var animal = item as? Dog {

animal.bark()

}else{

item.walk()

}

}

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

**Задание 6.2**

1. *Объявите класс (можно структуру) Bug, описывающую дефект в системе отслеживания ошибок. Он должен содержать следующие свойства: номер (*[*идентификатор*](https://ru.wikipedia.org/wiki/%D0%98%D0%B4%D0%B5%D0%BD%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%82%D0%BE%D1%80) *ID) дефекта (д.б. уникальным и последовательно изменяться при создании новой ошибки); короткое описание дефекта - Summary; описание шагов для выявления дефекта -Steps to Reproduce (опциональный массив String)*;  *кто сообщил о дефекте; дата и время, когда был обнаружен дефект; серьёзность (критичность) – Severity (возможные значения –* High, Medium, Low*) дефекта и*[*приоритет*](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BE%D1%80%D0%B8%D1%82%D0%B5%D1%82) *- Priority(возможные значения –* Blocker, Critical, Major, Minor *) решения (задайте на основе перечислений); кто ответственен за устранение дефекта (Assignee); текущее состояние* Status *(статус, возможные значения –* Open, Closed, Resolved, Reopened, In Progress *) дефекта (задайте на основе перечисления); версия продукта, в которой дефект исправлен (м.б. nil).*
2. *Предусмотрите методы инициализации, методы изменяющие приоритет решения, назначения ответственного за исправление дефекта, разрещения дефекта (изменяет стаус).*
3. *Напишите сабскрипт для возврата описания шага по номеру.*
4. *Задайте lazy свойство.*
5. *Сделайте одно из хранимых свойств наблюаемым.*