**Лабораторная работа №7**

**ПСЕВДОНИМЫ ANY И ANYOBJECT**

**Псевдонимы Any и AnyObject**

Swift предлагает два специальных псевдонима, позволяющих работать с неопределенными типами:

* AnyObject соответствует произвольному экземпляру любого класса;
* Any соответствует произвольному типу данных.

Данные псевдонимы позволяют корректно обрабатывать ситуации, когда конкретное наименование типа или класса неизвестно либо набор возможных типов может быть разнородным.

**Псевдоним Any**

Благодаря псевдониму Any можно создавать хранилища неопределенного типа данных. Объявим массив, который может содержать элементы произвольных типов

var things = [Any]()

things.append(0)

things.append(0.0)

things.append(42)

things.append("hello")

things.append((3.0, 5.0))

things.append({ (name: String) -> String in "Hello, \(name)" })

Массив things содержит значения типов: Int, Double, String, (Double, Double) и (String)->String . То есть перед вами целый набор различных типов данных. При запросе любого из элементов массива вы получите значение не того типа данных, который предполагался при установке конкретного значения, а типа Any.

ПРИМЕЧАНИЕ. Псевдоним Any не совместим с протоколом Hashable, поэтому использование типа Any там, где необходимо сопоставление, невозможно. Это относится, например, к ключам словарей.

**Приведение типа Any**

Для анализа каждого элемента массива необходимо выполнить приведение типа. Так вы сможете получить каждый элемент, преобразованный в его действительный тип данных.

**Псевдоним AnyObject**

Псевдоним AnyObject позволяет указать на то, что в данном месте должен или может находиться экземпляр любого класса. При этом вы будете довольно часто встречать массивы данного типа при разработке программ с использованием фреймворка Cocoa Foundation. Данный фреймворк написан на Objective-C, а этот язык не имеет массивов с явно указанными типами данных.

let someObjects: [AnyObject] = [Dog(),NoisyDog(),Dog()]

При использовании псевдонима AnyObject нет ограничений на использование классов только из одной иерархической структуры. В данном примере если вы извлекаете произвольный элемент массива, то получаете экземпляр класса AnyObject, не имеющий свойств и методов для взаимодействия с ним.

**Приведение типа AnyObject**

Порой вы точно знаете, что все элементы типа AnyObject на самом деле имеют некоторый определенный тип. В таком случае для анализа элементов типа AnyObject необходимо выполнить приведение типа

for object in someObjects {

let animal = object as! Dog

print("This is bad Dog")

}

for object in someObjects as! [Dog]{

print("This is bad Dog")

}

**Инициализаторы и деинициализаторы**

При реализации собственных типов данных во многих случаях не требуется создавать собственный инициализатор, так как классы и структуры имеют встроенные инициализаторы.

Инициализаторы класса и структуры, производящие установку значений свойств, называются назначенными (designated). Вы можете разработать произвольное количество назначенных инициализаторов с отличающимся набором параметров в пределах одного объектного типа. При этом должен существовать хотя бы один назначенный инициализатор, производящий установку значений всех свойств (если они существуют), и один из назначенных инициализаторов должен обязательно вызываться при создании экземпляра. Назначенный инициализатор не может вызывать другой назначенный инициализатор, то есть использование *конструкции self.init() запрещено*.

Единственный инициализатор, который может вызывать назначенный инициализатор, — это инициализатор производного класса, вызывающий инициализатор родительского класса для установки значений наследуемых свойств.

**Вспомогательные инициализаторы**

Помимо назначенных, в Swift существуют вспомогательные (convenience) инициализаторы. Они являются вторичными и поддерживающими. Вы можете определить вспомогательный инициализатор для проведения настроек и обязательного вызова одного из назначенных инициализаторов. Вспомогательные инициализаторы не являются обязательными для их реализации в типе. Создавайте их, если это обеспечивает наиболее рациональный путь решения поставленной задачи. Синтаксис объявления вспомогательных инициализаторов не слишком отличается от синтаксиса назначенных.

СИНТАКСИС

convenience init(параметры) {

// тело инициализатора

}

Вспомогательный инициализатор объявляется с помощью модификатора convenience, за которым следует ключевое слово init. Данный тип инициализатора также может принимать входные аргументы и устанавливать значения для свойств.

class Quadruped{

var name:String;

var type:String;

init(){

self.name="noname"

self.type=" "

}

}

class Dog: Quadruped {

override init(){

super.init()

self.type = "dog"

}

convenience init(text: String){

self.init()

print(text)

}

func bark(){

print("woof")

}

func printName(){

print(self.name)

}

}

var myDog = Dog(text: "Экземпляр класса Dog создан")

В результате при создании нового экземпляра класса Dog вам будет предложено выбрать один из двух инициализаторов: init() или init(text:). Вспомогательный инициализатор вызывает назначенный и выводит текст на консоль.

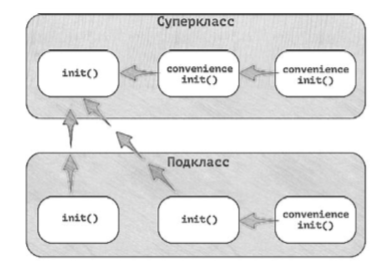
Наследование инициализаторов отличается от наследования обычных методов суперкласса. Есть два важнейших правила, которые необходимо помнить:

* Если подкласс имеет собственный назначенный инициализатор, то инициализаторы родительского класса не наследуются.
* Если подкласс переопределяет все назначенные инициализаторы суперкласса, то он наследует и все его вспомогательные инициализаторы.

**Отношения между инициализаторами**

В вопросах отношений между инициализаторами Swift соблюдает следующие правила:

* Назначенный инициализатор подкласса должен вызвать назначенный инициализатор суперкласса.
* Вспомогательный инциализатор должен вызвать назначенный инициализатор того же объектного типа.
* Вспомогательный инициализатор в конечном счете должен вызвать назначенный инициализатор.



Здесь изображен суперкласс с одним назначенным и двумя вспомогательными инициализаторами. Один из вспомогательных инициализаторов вызывает другой, который в свою очередь вызывает назначенный. Также изображен подкласс, имеющий два собственных назначенных инициализатора и один вспомогательный. Вызов любого инициализатора из изображенных должен в конечном итоге вызывать назначенный инициализатор суперкласса (левый верхний блок).

**Проваливающиеся инициализаторы**

В некоторых ситуациях бывает необходимо определить объектный тип, создание экземпляра которого может закончиться неудачей, вызванной некорректным набором внешних параметров, отсутствием какого-либо внешнего ресурса или иным обстоятельством. Для этой цели служат проваливающиеся (failable) инициализаторы. Они способны возвращать nil при попытке создания экземпляра. И это их основное предназначение.

СИНТАКСИС

init?(параметры) {

// тело инициализатора

}

Для создания проваливающегося инициализатора служит ключевое слово init? (со знаком вопроса), который говорит о том, что возвращаемый экземпляр будет опционалом или его не будет вовсе. В теле инициализатора должно присутствовать выражение return nil.

class Rectangle{

var height: Int

var weight: Int

init?(height h: Int, weight w: Int){

self.height = h

self.weight = w

if !(h > 0 && w > 0) {

return nil

}

}

}

var rectangle = Rectangle(height: 56, weight: -32) // возвращает nil

Инициализатор принимает и проверяет значения двух параметров. Если хотя бы одно из них не больше нуля, то возвращается nil. Обратите внимание на то, что, прежде чем вернуть nil, инициализатор устанавливает значения всех хранимых свойств.

ВНИМАНИЕ. В классах проваливающийся инициализатор может вернуть nil только после *установки значений всех хранимых свойств*. В случае структур данное ограничение отсутствует.

Назначенный инициализатор в подклассе может переопределить проваливающийся инициализатор суперкласса, а проваливающийся инициализатор может вызывать назначенный инициализатор того же класса. Не забывайте, что в случае использования проваливающегося инициализатора возвращается опционал. Поэтому прежде чем работать с экземпляром, необходимо выполнить извлечение опционального значения. Вы можете использовать проваливающийся инициализатор для выбора подходящего члена перечисления, основываясь на значениях входных аргументов.

Альтернативой инициализатору init? служит оператор init!. Разница в них заключается лишь в том, что второй возвращает неявно извлеченный экземпляр объектного типа, поскольку для работы с ним не требуется дополнительно извлекать опциональное значение. При этом все еще может возвращаться nil.

**Обязательные инициализаторы**

Обязательный (required) инициализатор — это инициализатор, который обязательно должен быть определен во всех подклассах данного класса.

СИНТАКСИС

required init(параметры) {

// тело инициализатора

}

Для объявления обязательного инициализатора перед ключевым словом init указывается модификатор required. Кроме того, модификатор required необходимо указывать перед каждой реализацией данного инициализатора в подклассах, чтобы последующие подклассы также реализовывали этот инициализатор.

При реализации инициализатора в подклассе ключевое слово override не используется.

**Деинициализаторы**

Деинициализаторы являются отличительной особенностью *классов*. Деинициализатор автоматически вызывается во время уничтожения экземпляра класса. Вы не можете вызвать деинициализатор самостоятельно. Один класс может иметь максимум один деинициализатор. С помощью деинициализатора вы можете, например, освободить используемые экземпляром ресурсы, вывести на консоль журнал или выполнить любые другие действия.

СИНТАКСИС

deinit {

// тело деинициализатора

}

Для объявления деинициализатора предназначено ключевое слово deinit, после которого в фигурных скобках указывается тело деинициализатора.

Деинициализатор суперкласса *наследуется подклассом* и вызывается автоматически в конце работы деинициализаторов подклассов. Деинициализатор суперкласса вызывается всегда, даже если деинициализатор подкласса отсутствует. Кроме того, экземпляр класса не удаляется, пока не закончит работу деинициализатор, поэтому все значения свойств экземпляра остаются доступными в теле деинициализатора.

class SuperClass {

init?(isNil: Bool){

if isNil == true {

return nil

}else{

print("Экземпляр создан")

}

}

deinit {

print("Деинициализатор суперкласса")

}

}

class SubClass:SuperClass{

deinit {

print("Деинициализатор подкласса")

}

}

var obj = SubClass(isNil: false)

obj = nil

//Экземпляр создан

//Деинициализатор подкласса

//Деинициализатор суперкласса

При создании экземпляра класса SubClass на консоль выводится соответствующее сообщение, так как данный функционал находится в наследуемом от суперкласса проваливающемся инициализаторе. В конце программы мы удаляем созданный экземпляр, передав ему в качестве значения nil. При этом вывод на консоль показывает, что первым выполняется деинициализатор подкласса, потом — суперкласса.

**Удаление экземпляров и ARC**

Любой созданный экземпляр объектного типа данных, как и вообще любое хранилище вашей программы, занимает некоторый объем оперативной памяти. Если не производить своевременное уничтожение экземпляров и освобождение занимаемых ими ресурсов, то в программе может произойти утечка памяти.

Одним из средств решения проблемы исключения утечек памяти в Swift является использование деинициализаторов, но возможности Swift не ограничиваются только этим.

**Уничтожение экземпляров**

Непосредственно перед уничтожением экземпляра класса вызывается деинициализатор, при этом остался нерассмотренным вопрос о том, как удаляется экземпляр. Любой экземпляр может быть удален одним из двух способов:

* его самостоятельно уничтожает разработчик;
* его уничтожает Swift.

**Область видимости**

Ранее мы уничтожали созданный экземпляр опционального типа SubClass?, передавая ему в качестве значения nil.

class myClass{

var description: String

init(description: String){

print("Экземпляр \(description) создан")

self.description = description

}

deinit{

print("Экземпляр \(self.description) уничтожен")

}

}

var myVar1 = myClass(description: "ОДИН")

if true {

var myVar2 = myClass(description: "ДВА")

}

//Консоль: Экземпляр ОДИН создан

//Экземпляр ДВА создан

//Экземпляр ДВА уничтожен

**Количество ссылок на экземпляр**

Рассмотрим пример, в котором на один экземпляр указывает несколько разных ссылок

class myClass{

var description: String

init(description: String){

print("Экземпляр \(description) создан")

self.description = description

}

deinit{

print("Экземпляр \(self.description) уничтожен")

}

}

var myVar1 = myClass(description: "ОДИН")

var myVar2 = myVar1

myVar1 = myClass(description: "ДВА")

myVar2 = myVar1

//Экземпляр ОДИН создан

//Экземпляр ДВА создан

//Экземпляр ОДИН уничтожен

В переменной myVar1 изначально хранится ссылка на экземпляр класса myClass. После записи данной ссылки в переменную myVar2 на созданный экземпляр уже указывают две ссылки. В результате этот экземпляр удаляется лишь тогда, когда удаляется последняя ссылка на него. Не забывайте, что экземпляры классов в Swift передаются не копированием, а по ссылке.

ПРИМЕЧАНИЕ. Экземпляр существует до тех пор, пока на него указывает хотя бы одна ссылка.

*Пример утечки памяти.* Рассмотрим ситуацию, при которой может возникнуть утечка памяти. Разработаем класс, который может описать человека и его родственные отношения с другими людьми. Для этого в качестве типа свойств класса будет указан сам тип

class Human {

let name: String

var child = [Human?]()

var father: Human?

var mother: Human?

init(name: String){

self.name = name

}

deinit {

print(self.name+" — удален")

}

}

if 1==1 {

var Kirill = Human(name: "Кирилл")

var Olga = Human(name: "Ольга")

var Aleks = Human(name: "Алексей")

Kirill.father = Aleks

Kirill.mother = Olga

Aleks.child.append(Kirill)

Olga.child.append(Kirill)

}

На консоль ничего не выводится, хотя все операции выполняются в теле условного оператора, то есть в ограниченной области видимости. Cоздано три экземпляра, указаны перекрестные ссылки друг на друга, но эти экземпляры вовремя не удаляются. Экземпляры остаются неудаленными, поскольку к моменту, когда их область видимости заканчивается, ссылки на объекты все еще существуют, и Swift не может понять, какие из ссылок можно удалить, а какие нельзя. Это типичный пример утечки памяти в приложениях.

**Сильные и слабые ссылки**

Для решения описанной ситуации, когда Swift не знает, какие из ссылок можно удалять, а какие нет, существует специальный механизм, называемый сильными и слабыми ссылками. Все создаваемые ссылки на экземпляр по умолчанию являются сильными. И когда два объекта указывают друг на друга сильными ссылками, то Swift не может принять решение о том, какую из ссылок можно удалить первой. Для решения проблемы некоторые ссылки можно преобразовать в слабые.

Слабые ссылки определяются с помощью ключевых слов weak и unowned. Модификатор weak указывает на то, что хранящаяся в параметре ссылка может быть в автоматическом режиме заменена на nil. Поэтому модификатор weak доступен только для опционалов. Но помимо опционалов бывают типы данных, которые обязывают переменную хранить значение (все неопциональные типы данных). Для создания слабых ссылок на неопционалы служит модификатор unowned.

class Human {

let name: String

var child = [Human?]()

weak var father: Human?

weak var mother: Human?

init(name: String){

self.name = name

}

deinit {

print(self.name+" — удален")

}

}

if 1==1 {

var Kirill = Human(name: "Кирилл")

var Olga = Human(name: "Ольга")

var Aleks = Human(name: "Алексей")

Kirill.father = Aleks

Kirill.mother = Olga

Aleks.child.append(Kirill)

Olga.child.append(Kirill)

}

//Алексей — удален

//Ольга — удален

//Кирилл — удален

В результате все три объекта будут удалены, так как после удаления слабых ссылок никаких перекрестных ссылок не остается. Указанные ключевые слова можно использовать *только для хранилища определенного экземпляра класса*. Например, вы не можете указать слабую ссылку на массив экземпляров или на кортеж, состоящий из экземпляров класса.

**Автоматический подсчет ссылок**

ARC (Automatic Reference Counting — автоматический подсчет ссылок)

На самом деле во всех наших действиях с экземплярами классов всегда участвовал механизм автоматического подсчета ссылок.

ARC в Swift автоматически управляет занимаемой памятью, удаляя неиспользуемые объекты. Например, на один и тот же экземпляр может указывать произвольное количество ссылок, и ARC в любой момент времени знает, сколько таких ссылок хранится в переменных, константах и свойствах. Как только последняя ссылка на экземпляр будет удалена или ее область видимости завершится, ARC незамедлительно вызовет деинициализатор и уничтожит объект.

**Сильные ссылки в замыканиях**

Сильные ссылки могут также стать источником проблем при их передаче в качестве входных параметров в замыкани. Захватываемые замыканиями экземпляры классов передаются по сильной ссылке и не освобождаются, когда замыкание уже не используется. Рассмотрим пример. Здесь создается пустое опциональное замыкание, которому в зоне ограниченной области видимости передается значение (тело замыкания).

class Human{

var name = "Человек"

deinit{

print("Объект удален")

}

}

var closure : (() -> ())?

if true{

var human = Human()

closure = {

print(human.name)

}

closure!()

}

print("Программа завершена")

//Консоль: Человек

//Программа завершена

Так как условный оператор ограничивает область видимости переменной human, содержащей экземпляр класса Human, то, казалось бы, данный объект должен быть удален вместе с окончанием условного оператора. Однако по выводу на консоль видно, что экземпляр создается, но перед завершением программы его деинициализатор не вызывается. Созданное опциональное замыкание использует сильную ссылку на созданный внутри условного оператора экземпляр класса. Так как замыкание является внешним по отношению к условному оператору, а ссылка сильной, Swift самостоятельно не может принять решение о возможности удаления ссылки и последующего уничтожения экземпляра. Для решения проблемы в замыкании необходимо выполнить захват переменной, указав при этом, что в переменной содержится слабая ссылка

class Human{

var name = "Человек"

deinit{

print("Объект удален")

}

}

var closure : (() -> ())?

if true{

var human = Human()

// измененное замыкание

closure = {

[unowned human] in

print(human.name)

}

closure!()

}

//Человек

//Объект удален

Захватываемый параметр human является локальным для замыкания и условного оператора, поэтому Swift без проблем может самостоятельно принять решение об уничтожении хранящейся в нем ссылки. В данном примере используется модификатор unowned, поскольку объектный тип не является опционалом.

**Опциональные цепочки**

Как вы знаете, опционалы могут содержать некоторое значение, а могут не содержать его вовсе. Для доступа к опциональным значениям мы выполняем их принудительное извлечение, указывая знак восклицания, что в случае несуществующего значения вызывает ошибку.

Но что делать, если в опциональном свойстве хранится экземпляр, к характеристикам которого требуется получить доступ? Рассмотрим пример, в котором два класса описывают некую персону и ее место жительства.

class Person {

var residence: Residence?

}

class Residence {

var numberOfRooms = 1

}

Экземпляры класса Person имеют единственное свойство со ссылкой на экземпляр класса Residence, который также имеет всего одно свойство. Если вы создаете новый экземпляр класса Person, то свойство residence имеет значение nil, поскольку оно является опционалом. Если вы попытаетесь получить доступ к свойству numberOfRooms, используя принудительное извлечение, то получите ошибку, так как значения не существует. Данный способ будет работать корректно только тогда, когда в свойстве residence хранится ссылка на экземпляр.

Для решения данной проблемы необходимо опциональное связывание.

var man = Person()

if let manResidence = man.residence {

print("Есть дом с \(manResidence.numberOfRooms) комнатами")

}else{

print("Нет дома")

}

// Нет дома

Представленный подход позволяет решить проблему, но потребует писать лишний код, если вложенность классов в качестве опциональных свойств окажется многоуровневой. Создадим новый класс, описывающий комнату, и добавим ссылку на экземпляр в класс Residence.

class Person {

var residence: Residence?

}

class Residence {

var numberOfRooms = 1

var room: Room?

}

class Room {

var square: Int = 10

}

Как видите, для доступа к свойству square требуется строить вложенные конструкции опционального связывания. Альтернативным способом доступа является использование опциональных цепочек. Они позволяют в одном выражении написать полный путь до требуемого элемента, при этом после каждого опционала необходимо ставить символ вопроса вместо восклицания. В примере создается трехуровневая структура вложенности, в которой требуется получить доступ к свойству room . При этом оба свойства (и residence, и room) — опционалы.

var man = Person()

var residence = Residence()

var room = Room()

man.residence = residence

residence.room = room

if let square = man.residence?.room?.square {

print("Площадь \(square) кв.м.")

}else{

print("Комнаты отсутствуют")

}

// Площадь 10 кв.м.

Если в каком-либо из узлов опциональной последовательности man. residence?.room?.square отсутствует значение (оно равно nil), то операция опционального связывания выполнена не будет и произойдет переход к альтернативной ветке условного оператора. В случае, когда опциональная цепочка не может получить доступ к элементу, результатом выражения является nil. Для проверки доступности элемента вы можете просто сравнить ведущую к нему опциональную цепочку с nil.

ПРИМЕЧАНИЕ Опциональную последовательность можно было использовать и в первом примере с двухуровневой вложенностью. Это также обеспечило бы более удобный способ доступа к свойству экземпляра. Вы можете использовать опциональные цепочки для вызова свойств, методов и сабскриптов для любого уровня вложенности типов друг в друга. Это позволяет «пробираться» через подсвойства внутри сложных моделей вложенных типов и проверять возможность доступа к свойствам, методам и сабскриптам этих подсвойств.

**Установка значений через опциональные цепочки**

Использование опциональных цепочек не ограничивается получением свойств и вызовом сабскриптов и методов. Они также могут быть использованы и для установки значений вложенных свойств. Вернемся к примеру с жилищем человека. Пусть нам необходимо изменить значение свойства square.

man.residence?.room?.square = 36

man.residence?.room?.square // Optional(36)

Для решения поставленной задачи используется опциональная цепочка man.residence?.room?.square, указывающая на требуемый элемент .

Если на каком-то из этапов экземпляр будет отсутствовать, программа не вызовет ошибку, а лишь не выполнит данное выражение.

**Доступ к методам через опциональные цепочки**

Как отмечалось ранее, опциональные цепочки могут быть использованы не только для доступа к свойствам, но и для вызова методов. В класс Residence добавим новый метод, который должен обеспечивать вывод информации о количестве комнат .

class Residence {

var numberOfRooms = 1

var room: Room?

func returnNumberOfRooms() ->Int {

return numberOfRooms

}

}

Для вызова данного метода требуется использовать опциональную последовательность.

man.residence?.returnNumberOfRooms()

// вывод количества комнат

принцип доступа к методу точно такой же, как к свойству.

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

**Задание 7.1 (связано с пред. Лаб. Работой)**

1. *Добавьте к классу Bug вспомогательный и проваливающийся инициализаторы.*
2. *Определите срециализированные виды дефектов (наследование)*
3. *Создайте массив типа Any, содержащий дефекты разного типа. Используя приведенение типа разберите объекты в массиве по типу.*