Практическая работа № 9

Инициализацияи деинициализация, наследование

Цель работы:

1. Изучить работу инциализаторов, научиться применять на практике.
2. Получить навыки использования деинциализаторов.
3. Создать консольное приложение, работа которого будет построена на использовании инициализаторов и наследование.

Краткие теоретические сведения

Инициализация

Классы, структуры и перечисления, однажды объявленные в Swift , инициализируются для подготовки экземпляра класса. Начальное значение инициализируется для сохраненного свойства, а также для новых экземпляров. Инициализируются значения, чтобы продолжить работу. Ключевое слово для создания функции инициализации выполняется методом init ().Инициализатор Swift отличается от Objective-C тем, что он не возвращает никаких значений. Его функция заключается в проверке инициализации вновь созданных экземпляров перед их обработкой. Swift также предоставляет процесс «деинициализации» для выполнения операций управления памятью после освобождения экземпляров.

## Роль инициализатора для сохраненных свойств

Сохраненное свойство должно инициализировать экземпляры для своих классов и структур перед обработкой экземпляров. Хранимые свойства используют инициализатор для назначения и инициализации значений, тем самым устраняя необходимость вызова наблюдателей свойств.Инициализатор используется в хранимом свойстве

* Создать начальное значение.
* Чтобы назначить значение свойства по умолчанию в определении свойства.
* Для инициализации экземпляра для определенного типа данных используется init (). Внутри функции init () аргументы не передаются.

### Синтаксис

init() {

//New Instance initialization goes here

}

### Пример

*struct rectangle {*

*var length:Double*

*var breadth:Double*

*init(){*

*length =6*

*breadth =12*

*}*

*}*

*var area = rectangle()*

*print("area of rectangle is \(area.length\*area.breadth)")*

Когда мы запускаем вышеуказанную программу, используя площадку, мы получаем следующий результат –

*area of rectangle is 72.0*

Здесь структура ‘rectangle’ инициализируется с длиной и шириной элементов как ‘Double’ типы данных. Метод Init () используется для инициализации значений для вновь создаваемых элементов длины и двойника. Площадь прямоугольника вычисляется и возвращается путем вызова функции прямоугольника.

## Установка значений свойств по умолчанию

Язык Swift предоставляет функцию Init () для инициализации значений сохраненных свойств.Кроме того, пользователь может инициализировать значения свойств по умолчанию при объявлении членов класса или структуры. Когда свойство принимает одно и то же значение во всей программе, мы можем объявить его только в разделе объявления, а не инициализировать его в init (). Установка значений свойств по умолчанию позволяет пользователю, когда наследование определено для классов или структур.

*struct rectangle {*

*var length =6*

*var breadth =12*

*}*

*var area = rectangle()*

*print("area of rectangle is \(area.length\*area.breadth)")*

Когда мы запускаем вышеуказанную программу, используя площадку, мы получаем следующий результат –

*area of rectangle is 72*

Здесь вместо объявления длины и ширины в init () значения инициализируются в самом объявлении.

## Инициализация параметров

В языке Swift пользователь имеет возможность инициализировать параметры как часть определения инициализатора с помощью init ().

*structRectangle{*

*var length:Double*

*var breadth:Double*

*var area:Double*

*init(fromLength length:Double, fromBreadth breadth:Double){*

*self.length = length*

*self.breadth = breadth*

*area = length \* breadth*

*}*

*init(fromLeng leng:Double, fromBread bread:Double){*

*self.length = leng*

*self.breadth = bread*

*area = leng \* bread*

*}*

*}*

*let ar =Rectangle(fromLength:6, fromBreadth:12)*

*print("area is: \(ar.area)")*

*let are =Rectangle(fromLeng:36, fromBread:12)*

*print("area is: \(are.area)")*

Когда мы запускаем вышеуказанную программу, используя площадку, мы получаем следующий результат –

*area is: 72.0*

*area is: 432.0*

## Локальные и внешние параметры

Параметры инициализации имеют как локальные, так и глобальные имена параметров, аналогичные параметрам функций и методов. Объявление локального параметра используется для доступа в теле инициализации, а объявление внешнего параметра используется для вызова инициализатора. Инициализаторы Swift отличаются от инициализатора функций и методов тем, что не определяют, какой инициализатор используется для вызова каких функций.

Чтобы преодолеть это, Swift вводит автоматическое внешнее имя для каждого параметра в init ().Это автоматическое внешнее имя эквивалентно локальному имени, записанному перед каждым параметром инициализации.

*structDays{*

*let sunday, monday, tuesday:Int*

*init(sunday:Int, monday:Int, tuesday:Int){*

*self.sunday = sunday*

*self.monday = monday*

*self.tuesday = tuesday*

*}*

*init(daysofaweek:Int){*

*sunday = daysofaweek*

*monday = daysofaweek*

*tuesday = daysofaweek*

*}*

*}*

*let week =Days(sunday:1, monday:2, tuesday:3)*

*print("Days of a Week is: \(week.sunday)")*

*print("Days of a Week is: \(week.monday)")*

*print("Days of a Week is: \(week.tuesday)")*

*let weekdays =Days(daysofaweek:4)*

*print("Days of a Week is: \(weekdays.sunday)")*

*print("Days of a Week is: \(weekdays.monday)")*

*print("Days of a Week is: \(weekdays.tuesday)")*

Когда мы запускаем вышеуказанную программу, используя площадку, мы получаем следующий результат –

*Days of a Week is: 1*

*Days of a Week is: 2*

*Days of a Week is: 3*

*Days of a Week is: 4*

*Days of a Week is: 4*

*Days of a Week is: 4*

## Параметры без внешних имен

Когда внешнее имя не требуется для инициализации, подчеркивание ‘\_’ используется для переопределения поведения по умолчанию.

*structRectangle{*

*var length:Double*

*init(frombreadth breadth:Double){*

*length = breadth \*10*

*}*

*init(frombre bre:Double){*

*length = bre \*30*

*}*

*init(\_ area:Double){*

*length = area*

*}*

*}*

*let rectarea =Rectangle(180.0)*

*print("area is: \(rectarea.length)")*

*let rearea =Rectangle(370.0)*

*print("area is: \(rearea.length)")*

*let recarea =Rectangle(110.0)*

*print("area is: \(recarea.length)")*

Когда мы запускаем вышеуказанную программу, используя площадку, мы получаем следующий результат –

*area is: 180.0*

*area is: 370.0*

*area is: 110.0*

## Необязательные типы недвижимости

Когда хранимое свойство в каком-то экземпляре не возвращает никакого значения, это свойство объявляется с необязательным типом, указывающим, что для этого конкретного типа не возвращается «никакого значения». Когда сохраненное свойство объявляется как «необязательное», оно автоматически инициализирует значение «nil» во время самой инициализации.

*structRectangle{*

*var length:Double?*

*init(frombreadth breadth:Double){*

*length = breadth \*10*

*}*

*init(frombre bre:Double){*

*length = bre \*30*

*}*

*init(\_ area:Double){*

*length = area*

*}*

*}*

*let rectarea =Rectangle(180.0)*

*print("area is: \(rectarea.length)")*

*let rearea =Rectangle(370.0)*

*print("area is: \(rearea.length)")*

*let recarea =Rectangle(110.0)*

*print("area is: \(recarea.length)")*

Когда мы запускаем вышеуказанную программу, используя площадку, мы получаем следующий результат –

*area is: Optional(180.0)*

*area is: Optional(370.0)*

*area is: Optional(110.0)*

## Изменение свойств константы во время инициализации

Инициализация также позволяет пользователю изменять значение постоянного свойства. Во время инициализации свойство класса позволяет его экземплярам класса изменяться суперклассом, а не подклассом. Рассмотрим, к примеру, в предыдущей программе «длина» объявлена ​​как «переменная» в основном классе. Приведенная ниже программная переменная «length» модифицируется как «постоянная» переменная.

*structRectangle{*

*let length:Double?*

*init(frombreadth breadth:Double){*

*length = breadth \*10*

*}*

*init(frombre bre:Double){*

*length = bre \*30*

*}*

*init(\_ area:Double){*

*length = area*

*}*

*}*

*let rectarea =Rectangle(180.0)*

*print("area is: \(rectarea.length)")*

*let rearea =Rectangle(370.0)*

*print("area is: \(rearea.length)")*

*let recarea =Rectangle(110.0)*

*print("area is: \(recarea.length)")*

Когда мы запускаем вышеуказанную программу, используя площадку, мы получаем следующий результат –

*area is: Optional(180.0)*

*area is: Optional(370.0)*

*area is: Optional(110.0)*

## Инициализаторы по умолчанию

Инициализаторы по умолчанию предоставляют новый экземпляр всем его объявленным свойствам базового класса или структуры со значениями по умолчанию.

*class defaultexample {*

*var studname:String?*

*var stmark =98*

*varpass=true*

*}*

*var result = defaultexample()*

*print("result is: \(result.studname)")*

*print("result is: \(result.stmark)")*

*print("result is: \(result.pass)")*

Когда мы запускаем вышеуказанную программу, используя площадку, мы получаем следующий результат –

*result is: nil*

*result is: 98*

*result is: true*

Вышеуказанная программа определена с именем класса как «defaultexample». Три функции-члена по умолчанию инициализируются как ‘studname?’ хранить значения ‘nil’, ‘stmark’ как 98 и ‘pass’ как логическое значение ‘true’. Аналогично значения членов в классе могут быть инициализированы как значения по умолчанию перед обработкой типов членов класса.

## Назначенные и вспомогательные инициализаторы в действии

Основной (базовый) класс называется Food, который имеет одно простое свойство типа String, называемое name, и обеспечивает два инициализатора для создания экземпляров класса Food:

class Food {

var name: String

init(name: String) {

self.name = name

}

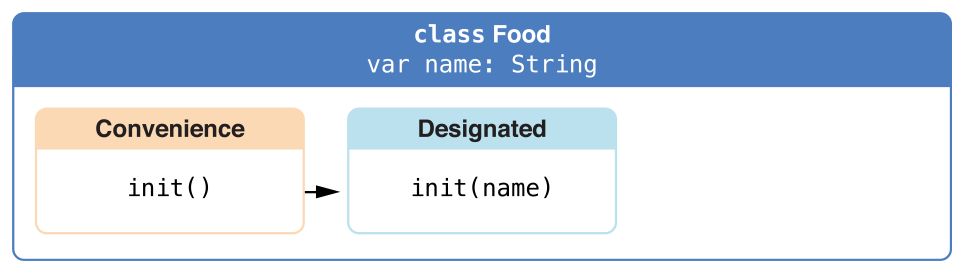
convenience init() {

self.init(name: "[Unnamed]")

}

}

Схема ниже показывает цепочку работы инициализаторов в классе Food :



Классы по умолчанию не имеют почленного инициализатора, так что класс Food предоставляет назначенный инициализатор, который принимает единственный аргумент name. Этот инициализатор может быть использован для создания экземпляра Food со специфичным именем:

let namedMeat = Food(name: "Бекон")

//имя namedMeat является "Бекон"

Инициализатор init(name: String) из класса Food, представлен в виде *назначенного*инициализатора, потому что он проверяет, что все хранимые свойства нового экземпляра Food полностью инициализированы. Класс Food не имеет суперкласса, так что инициализатор init(name: String) не имеет вызова super.init() для завершения своей инициализации.

Класс Food так же обеспечивает вспомогательный инициализатор init() без аргументов. Инициализатор init() предоставляет имя плейсхолдера для новой еды, делегируя к параметру name инициализатора init(name: String), давая ему значение [Unnamed] :

let mysteryMeat = Food()

//mysteryMeat называется "[Unnamed]"

Второй класс в иерархии - это подкласс RecipeIngredient класса Food. Класс RecipeIngredient создает модель ингредиентов в рецепте. Он представляет свойство quantity типа Int (в дополнение к свойству name, унаследованное от Food) и определяет два инициализатора для создания экземпляров RecipeIngredient :

classRecipeIngredient: Food{

var quantity: Int

init(name: String, quantity: Int) {

self.quantity = quantity

super.init(name: name)

}

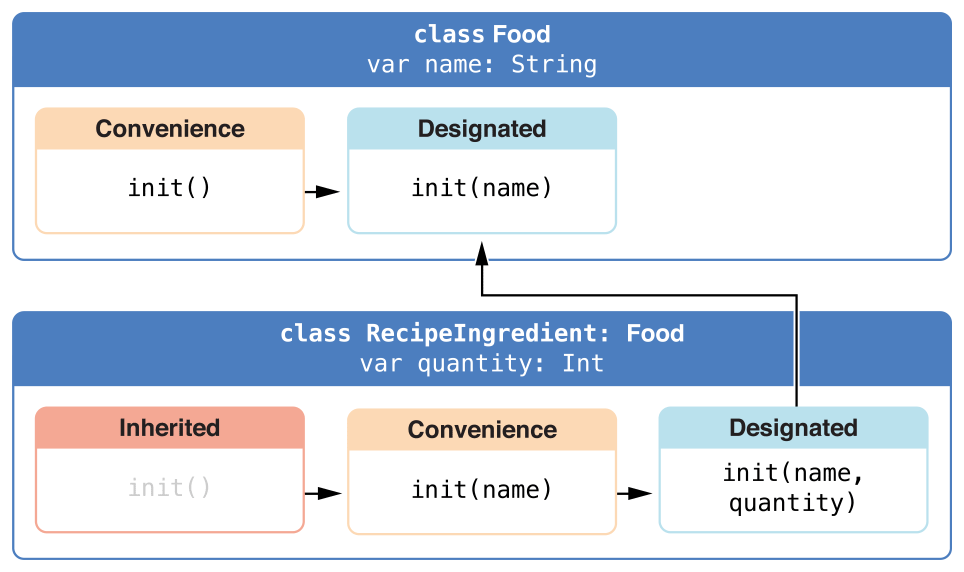
overrideconvenienceinit(name: String) {

self.init(name: name, quantity: 1)

}

}

Схема ниже показывает цепочку инициализаторов для класса RecipeIngredient:



Все три инициализатора могут быть использованы для создания новых RecipeIngredient экземпляров:

let oneMysteryItem = RecipeIngredient()

let oneBacon = RecipeIngredient(name: "Bacon")

let sixEggs = RecipeIngredient(name: "Eggs", quantity: 6)

Пример

|  |
| --- |
| classUser {        varage: Int      varname: String        init(){            age = 22          name = "Tom"      }      init(name: String, age: Int){            self.age = age          self.name = name      }        funcgetUserInfo(){            print("Имя: \(self.name); возраст: \(self.age)")      }  }    varbob: User = User(name: "Bob", age: 34)  bob.getUserInfo()   // Имя: Bob; возраст: 34 |

Второй инициализатор принимает два параметра name и age для установки свойств класса. Так как параметры и свойства класса называются одинаково, то для их разграничения вместе с названиями свойств используется ключевое слово **self**.

И при создании объекта в данном случае используется второй инициализатор: var bob: User = User(name: "Bob", age: 34)

Инициализаторы могут определять значения по умолчанию для параметров. Так, пример выше мы могли бы сократить следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | classUser {        varage: Int      varname: String        init(name: String= "Tom", age: Int= 22){            self.age = age          self.name = name      }        funcgetUserInfo(){            print("Имя: \(self.name); возраст: \(self.age)")      }  }  vartom = User()  tom.getUserInfo()   // Имя: Tom; возраст: 22 |

**Делегирование инициализации**

Одни инициализаторы могут вызывать другие. Вызывающие инициализаторы должны быть определены с ключевым словом **convenience**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | classUser {        varage: Int      varname: String        convenienceinit(){            self.init(name: "Tom", age: 22)      }      init(name: String, age: Int){            self.age = age          self.name = name      }        funcgetUserInfo(){            print("Имя: \(self.name); возраст: \(self.age)")      }  }    vartom: User = User()  tom.getUserInfo()   // Имя: Tom; возраст: 22 |

**Failable-инициализаторы**

Специальная разновидность инициализаторов (Failable Initializer) позволяет возвратить значение nil, если в процессе инициализации объекта произошла какая-нибудь ошибка. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | Class User{        Var name: String      Var age: Int      init?(name: String, age: Int){            self.name = name          self.age = age          if(age < 0){              returnnil          }      }  }  Var bob: User = User(name: "Bob", age: 34)!  print(bob.name) // Bob |

Поскольку пользователь, представленный классом User, в принципе не может иметь возраст меньше нуля. Поэтому ситуация, когда для возраста передается число меньше нуля, может рассматриваться как ошибочная. И в этом случае мы как раз можем использовать failable-инициализатор.

Для определения такого инициализатора после слова init ставится знак вопроса, а в самом инициализаторе можно предусмотреть ситуацию, при которой он возвращает значение **nil**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | init?(name: String, age: Int){        self.name = name      self.age = age      if(age < 0){          returnnil      }  } |

Возвращая nil, мы тем самым указываем, что мы не можем создать объект User по тем данным, которые переданы в инициализатор.

Важно учитывать, что объект, создаваемый этим инициализатором, будет представлять не тип User, а тип User?. Поэтому для получения значения нам надо еще использовать операцию ! (восклицательный знак):

|  |  |
| --- | --- |
| 1 | varbob: User = User(name: "Bob", age: 34)! |

Либо мы можем напрямую работать с объектом User?:

|  |  |
| --- | --- |
| 1 | varbob: User? = User(name: "Bob", age: 34) |

## Поэлементные инициализаторы для типов структуры

Когда пользовательские инициализаторы не предоставлены пользователем, типы структуры в Swift автоматически получат «членский инициализатор». Его основная функция заключается в инициализации новых экземпляров структуры по умолчанию для каждого элемента, а затем свойства нового экземпляра передаются элементному элементу инициализации по имени.

*structRectangle{*

*var length =100.0, breadth =200.0*

*}*

*let area =Rectangle(length:24.0, breadth:32.0)*

*print("Area of rectangle is: \(area.length)")*

*print("Area of rectangle is: \(area.breadth)")*

Когда мы запускаем вышеуказанную программу, используя площадку, мы получаем следующий результат –

*Area of rectangle is: 24.0*

*Area of rectangle is: 32.0*

Структуры инициализируются по умолчанию для их функций членства во время инициализации для «length» как «100.0» и «width» как «200.0». Но значения переопределяются при обработке переменных длины и ширины как 24.0 и 32.0.

## Делегирование инициализатора для типов значений

Делегирование инициализатора определяется как вызов инициализаторов из других инициализаторов. Его основная функция заключается в возможности повторного использования, чтобы избежать дублирования кода между несколькими инициализаторами.

*structStmark{*

*var mark1 =0.0, mark2 =0.0*

*}*

*struct stdb {*

*var m1 =0.0, m2 =0.0*

*}*

*struct block {*

*var average = stdb()*

*var result =Stmark()*

*init(){}*

*init(average: stdb, result:Stmark){*

*self.average = average*

*self.result = result*

*}*

*init(avg: stdb, result:Stmark){*

*let tot = avg.m1 -(result.mark1 /2)*

*let tot1 = avg.m2 -(result.mark2 /2)*

*self.init(average: stdb(m1: tot, m2: tot1), result: result)*

*}*

*}*

*let set1 = block()*

*print("student result is: \(set1.average.m1, set1.average.m2)*

*\(set1.result.mark1, set1.result.mark2)")*

*let set2 = block(average: stdb(m1:2.0, m2:2.0),*

*result:Stmark(mark1:5.0, mark2:5.0))*

*print("student result is: \(set2.average.m1, set2.average.m2)*

*\(set2.result.mark1, set2.result.mark2)")*

*let set3 = block(avg: stdb(m1:4.0, m2:4.0),*

*result:Stmark(mark1:3.0, mark2:3.0))*

*print("student result is: \(set3.average.m1, set3.average.m2)*

*\(set3.result.mark1, set3.result.mark2)")*

Когда мы запускаем вышеуказанную программу, используя площадку, мы получаем следующий результат –

*(0.0,0.0) (0.0,0.0)*

*(2.0,2.0) 5.0,5.0)*

*(2.5,2.5) (3.0,3.0)*

### Правила для делегирования инициализатора

|  |  |
| --- | --- |
| **Типы значений** | **Типы классов** |
| Наследование не поддерживается для типов значений, таких как структуры и перечисления. Ссылка на другие инициализаторы осуществляется через self.init | Наследование поддерживается.Проверяет, что все сохраненные значения свойств инициализированы |

## Наследование и инициализация класса

Типы классов имеют два вида инициализаторов, чтобы проверить, получают ли определенные сохраненные свойства начальное значение, а именно назначенные инициализаторы и удобные инициализаторы.

### Назначенные инициализаторы и удобные инициализаторы

|  |  |
| --- | --- |
| **Назначенный инициализатор** | **Удобный инициализатор** |
| Считается первичной инициализацией для класса | Рассматривается как поддерживающая инициализация для класса |
| Все свойства класса инициализируются, и для дальнейшей инициализации вызывается соответствующий инициализатор суперкласса. | Назначенный инициализатор вызывается с удобным инициализатором для создания экземпляра класса для конкретного варианта использования или типа входного значения |
| По крайней мере один назначенный инициализатор определен для каждого класса | Нет необходимости в обязательном определении удобных инициализаторов, когда класс не требует инициализаторов. |
| Init (параметры) {заявления} | удобство init (параметры) {заявления} |

### Программа для назначенных инициализаторов

class mainClass {

*var no1 :Int// local storage*

*init(no1 :Int){*

*self.no1 = no1 // initialization*

*}*

*}*

*classubclass: mainClass {*

*var no2 :Int// new subclass storage*

*init(no1 :Int, no2 :Int){*

*self.no2 = no2 // initialization*

*super.init(no1:no1)// redirect to superclass*

*}*

*}*

*let res = mainClass(no1:10)*

*letprint=ubclass(no1:10, no2:20)*

*print(“res is: \(res.no1)”)*

*print(“res is: \(print.no1)”)*

*print(“res is: \(print.no2)”)*

Когда мы запускаем вышеуказанную программу, используя площадку, мы получаем следующий результат –

*res is: 10*

*res is: 10*

*res is: 20*

### Программа для удобных инициализаторов

*classmainClass{*

*var no1 :Int// local storage*

*init(no1 :Int){*

*self.no1 = no1 // initialization*

*}*

*}*

*classubclass: mainClass {*

*var no2 :Int*

*init(no1 :Int, no2 :Int){*

*self.no2 = no2*

*super.init(no1:no1)*

*}*

*// Requires only one parameter for convenient method*

*override convenience init(no1:Int){*

*self.init(no1:no1, no2:0)*

*}*

*}*

*let res = mainClass(no1:20)*

*letprint=ubclass(no1:30, no2:50)*

*print(“res is: \(res.no1)”)*

*print(“res is: \(print.no1)”)*

*print(“res is: \(print.no2)”)*

Когда мы запускаем вышеуказанную программу, используя площадку, мы получаем следующий результат –

*res is: 20*

*res is: 30*

*res is: 50*

## Инициализация наследования и переопределение

Swift не позволяет своим подклассам наследовать инициализаторы суперкласса для их типов членов по умолчанию. Наследование применимо к инициализаторам суперкласса только в некоторой степени, что будет обсуждаться в разделе «Автоматическое наследование инициализатора».

Когда пользователю необходимо определить инициализаторы в суперклассе, подкласс с инициализаторами должен быть определен пользователем как пользовательская реализация. Когда переопределение должно выполняться подклассом, необходимо объявить ключевое слово «переопределение» суперкласса.

*Class sides {*

*var corners =4*

*var description:String{*

*return“\(corners) sides”*

*}*

*}*

*let rectangle = sides()*

*print(“Rectangle: \(rectangle.description)”)*

*class pentagon: sides {*

*override init(){*

*super.init()*

*corners =5*

*}*

*}*

*let bicycle = pentagon()*

*print(“Pentagon: \(bicycle.description)”)*

Когда мы запускаем вышеуказанную программу, используя площадку, мы получаем следующий результат –

*Rectangle: 4 sides*

*Pentagon: 5 sides*

## Требуемые инициализаторы

Чтобы объявить каждый подкласс инициализируемого ключевого слова ‘required’, необходимо определить его перед функцией init ().

*Class classA {*

*required init(){*

*var a =10*

*print(a)*

*}*

*}*

*class classB: classA {*

*required init(){*

*var b =30*

*print(b)*

*}*

*}*

*let res = classA()*

*letprint= classB()*

Деинициализация

Перед тем, как экземпляр класса должен быть освобожден, должен быть вызван deinitializer для освобождения пространства памяти. Ключевое слово deinit используется для освобождения областей памяти, занимаемых системными ресурсами. Деинициализация доступна только для типов классов.

## Деинициализация для освобождения пространства памяти

Swift автоматически освобождает ваши экземпляры, когда они больше не нужны, чтобы освободить ресурсы. Swift выполняет управление памятью экземпляров с помощью автоматического подсчета ссылок (ARC), как описано в разделе «Автоматический подсчет ссылок». Как правило, вам не нужно выполнять ручную очистку, когда ваши экземпляры освобождаются. Однако, когда вы работаете с собственными ресурсами, вам может потребоваться выполнить дополнительную очистку самостоятельно. Например, если вы создаете пользовательский класс, чтобы открыть файл и записать в него некоторые данные, вам может потребоваться закрыть файл до освобождения экземпляра класса.

Var counter =0;// for reference counting

class baseclass {

init(){

counter++;

}

deinit {

counter--;

}

}

varprint: baseclass?= baseclass()

print(counter)

print=nil

print(counter)

Когда мы запускаем вышеуказанную программу, используя площадку, мы получаем следующий результат –

1

0

Задание для самостоятельного выполнения

1. Создать класс по варианту, сделать назначенные и удобные инициализаторы, деинициализаторы.
2. Создать новый класс на основе исходного с учетом варианта задания (автопарк, библиотека, каталог, команда, группа, реестр фирм, склад компьютеров, спортклуб, автостоянка, каталог журнальных статей), использовать наследование.
3. Заполнить информацию и вывести на экран

Варианты

1. Автобус (номер маршрута, название начальной и конечной остановки, количество остановок по маршруту)
2. Книга (название книги, автор, издательство, количество страниц)
3. Файл (имя, время создания, дата создания, размер)
4. Спортсмен (ФИО, вид спорта, сколько лет занимается)
5. Студент (ФИО, группа, средний балл зачетки)
6. Фирма (название, вид услуг, годовой оборот)
7. Компьютер (производитель, частота процессора, количество ядер, объем памяти)
8. Спорт (вид спорта, количество занимающихся в городе, средняя стоимость комплекта инвентаря)
9. Автомобиль (марка, фирма-производитель,год выпуска, пробег)
10. Статья (название, автор, название журнала, год и месяц издания)