GeekBrains

ARC и управление памятью

Основы языка Swift

Оглавление

Введение

На прошлой лекции мы узнали про протоколы, расширения и дженерики. В рамках этой лекции вы познакомитесь с тем, что такое автоматический подсчет ссылок и как можно управлять памятью, чтобы избежать ее утечки.

На этой лекции вы узнаете про:

- Автоматический подсчет ссылок
- Виды ссылок
- Reference и value type
- Список захвата у замыканий
- is и as

Термины, используемые в лекции

Автоматический подсчет ссылок (или ARC - automatic reference counting) - это механизм для отслеживания и управления памятью в приложении.

Reverence и value type

Мы уже изучили классы и структуры, поэтому пришло время узнать об одном из главных отличий классов от структур. Они относятся к разным типами и в памяти хранятся по разному.

В Swift типы делятся на две категории: ссылочный тип (reference type) и тип значения (value type).

Тип значения - каждый экземпляр хранит уникальную копию своих данных, то есть при присвоении нового значения переменной происходит создание новой копии(экземпляра).

Ссылочный тип - каждый экземпляр использует одну копию данных, то есть при присвоении значения переменной сохраняется ссылка на тот же экземпляр.

К reference type относятся замыкания и классы, а к value type все остальное, например, перечисления, структуры, кортежи.

Кроме то, ссылочный тип хранится в куче (используется для распределения динамической памяти), а тип значения в стеке (используется для распределения статической памяти)

Отличие reference и value type

reference и value type отличаются своим поведением в различных ситуациях, далее мы рассмотрим случаи, когда классы и структуры могут повести себя по разному

Копирование значений

Когда мы передаем значение в константу или переменную ссылочного типа, то передается ссылка на область в памяти, в которой хранится этот объект, а когда мы передаем тип значения, то происходит копирование этого значения.

Чтобы лучше понять, давайте рассмотрим следующую ситуацию: у нас есть класс Cafe, добавим в него еще название, то есть переменную name

```
class Cafe: CafeProtocol {
    var coffee: [Coffee]
    var tea: [Tea]
    var name: String

    required init(coffee: [Coffee], tea: [Tea], name: String) {
        self.coffee = coffee
        self.tea = tea
        self.name = name
    }
```

```
    convenience init(name: String) {
        let latte = Coffee(name: "Latte", isSugar: true, isIce:
false, size: .l)
        let cappuccino = Coffee(name: "Cappuccino", isSugar:
false, isIce: false, size: .m)
        let blackTea = Tea(name: "Black Tea", cost: 50)
        self.init(coffee: [latte, cappuccino], tea: [blackTea],
name: name)
    }

    func addCoffee(coffee: Coffee) {
        self.coffee.append(coffee)
    }

    func printDrink<T>(drink: T) {
        print(drink)
    }
}
```

Теперь создадим экземпляр класса Cafe, затем присвоим значение этой переменной в новую переменную

```
var cafe = Cafe(name: "Cafe")
var newCafe = cafe
```

И cafe, и newCafe теперь указывают на один и тот же объект в памяти, поэтому, если мы изменим название в newCafe, то название cafe тоже изменится.

```
95  var cafe = Cafe(name: "Cafe")
96  var newCafe = cafe
97  newCafe.name = "New cafe"
98
99  print("Название cafe \(cafe.name)")
100 print("Название newCafe \(newCafe.name)")
```

```
Название cafe New cafe
Название newCafe New cafe
```

Как мы видим, после того, как мы изменили название newCafe, название cafe тоже изменилось на “New cafe”

Теперь же рассмотрим структуру Coffee

```
struct Coffee {
    enum CoffeeSize {
        case s
        case m
        case l
    }

    var name: String
    var isSugar: Bool
    var isIce: Bool
    var cost: Double = 110
    var size: CoffeeSize
}
```

Давайте создадим переменную latte и потом присвоим ее значение новой переменной, а именно newLatte

```
var latte = Coffee(name: "Latte", isSugar: true, isIce: false,
size: .l)
var newLatte = latte
```

Структура имеет тип значения, поэтому, когда мы в переменную newLatte кладем значение переменной latte, происходит копирование значение, поэтому, если мы изменим свойство name в newLatte, значение name в latte останется прежним.

```
33 var latte = Coffee(name: "Latte", isSugar: true, isIce: false, size:
       .l)
34 var newLatte = latte
35 newLatte.name = "Super new latte"
36
37 print("Название latte \(latte.name)")
38 print("Название newLatte \(newLatte.name)")
39
```

```
Название latte Latte
Название newLatte Super new latte
```

Изменение константы

У нас есть класс Cafe и структура Coffee, и там, и там есть свойство name, которое является переменной. Теперь создадим две константы: cafe и latte

```
let cafe = Cafe(name: "Cafe")
let latte = Coffee(name: "Latte", isSugar: true, isIce: false,
size: .l)
```

cafe это класс, а coffee - структура, попробуем изменить название и кафе и кофе

```
cafe.name = "New cafe"
latte.name = "New latte"
```

Cannot assign to property: 'latte' is a 'let' constant

Change 'let' to 'var' to make it mutable Fix

В случае с классом никаких проблем не возникло, а вот при изменении свойства в структуре сразу возникает ошибка, так как при изменении свойства структуры полностью меняется объект этой самой структуры, что недопустимо для констант.

Оператор идентичности

Для сравнения экземпляров класса в Swift используется оператор идентичности “===” или неидентичности “!==”. Давайте создадим переменную cafe, затем положим значение этой переменной в newCafe.

```
var cafe = Cafe(name: "Cafe")
var newCafe = cafe
```

Теперь создадим еще одну переменную superNewCafe, она будет точно такой же, как cafe

```
var superNewCafe = Cafe(name: "Cafe")
```

Начнем сравнивать экземпляры объектов

```
108
109  print(cafe === newCafe)
```

```
true
```

В данном случае будет выведено true, так как объекты ссылаются на одну и ту же область памяти

А вот если сравнить cafe и superNewCafe, то будет выведено false, так как, хоть эти объекты и полностью идентичны, они ссылаются на разные области памяти

```
108
109  print(cafe === superNewCafe)
110
```

```
false
```

К структурам оператор идентичности неприменим, если мы попробуем его использовать на двух структурах - выскочит ошибка

```
print(latte === newLatte)
```

Argument type 'Coffee' expected to be an instance of a class or class-constrained type

Автоматический подсчет ссылок

Мы узнали про ссылочный тип, понимаем, что он связан с какими-то ссылками, но что это за ссылки и зачем они нужны? Тут мы плавно перешли к автоматическому подсчету ссылок.

Для отслеживания и управления памятью в Swift используется автоматический подсчет ссылок (ARC), он автоматически освобождает память, которая используется экземплярами класса, если они в ней больше не нуждаются. Когда вы создаете новый экземпляр класса, автоматический подсчет ссылок выделяет кусок памяти для хранения информации об этом самом классе. Когда же экземпляр больше не нужен, ARC освобождает память, которая используется экземпляром класса.

Как работает ARC

Чтобы лучше разобраться в том, как работает автоматический подсчет ссылок, вернемся к классу Cafe. Давайте для начала добавим в инициализатор print, когда инициализация вызвана.

```
class Cafe: CafeProtocol {
    var coffee: [Coffee]
    var tea: [Tea]
    var name: String

    required init(coffee: [Coffee], tea: [Tea], name: String) {
        self.coffee = coffee
        self.tea = tea
        self.name = name
```

```
        print("Инициализация")
    }

    convenience init(name: String) {
        let latte = Coffee(name: "Latte", isSugar: true, isIce:
false, size: .l)
        let cappuccino = Coffee(name: "Cappuccino", isSugar:
false, isIce: false, size: .m)
        let blackTea = Tea(name: "Black Tea", cost: 50)
        self.init(coffee: [latte, cappuccino], tea: [blackTea],
name: name)
    }

    func addCoffee(coffee: Coffee) {
        self.coffee.append(coffee)
    }

    func printDrink<T>(drink: T) {
        print(drink)
    }
}
```

Теперь добавим деинициализатор в класс Cafe, он будет вызван, когда экземпляр должен будет быть освобожден. В deinit вызовем print.

```
deinit {
        print("Деинициализация")
    }
```

Далее создадим три переменных типа Cafe, которые также будут опционалами.

```
var cafe: Cafe?
var newCafe: Cafe?
var superNewCafe: Cafe?
```

Так как переменные являются опционалами, они автоматически были проинициализированы со значением nil. В данный момент они пока не ссылаются на Cafe, поэтому количество ссылок на Cafe равно 0.

Теперь создадим новый экземпляр и присвоим его одной из переменных

```
cafe = Cafe(name: "Cafe")
```

Запустим код и увидим в консоль “Инициализация”, теперь есть первая сильная ссылка на экземпляр Cafe, а значит ARC гарантирует, что экземпляр Cafe хранится в памяти и не будет освобожден.

```
123  cafe = Cafe(name: "Cafe")
```

Инициализация

Теперь назначим переменным newCafe и superNewCafe значение переменной cafe

```
cafe = Cafe(name: "Cafe")
newCafe = cafe
superNewCafe = cafe
```

В данный момент появилось уже 3 сильных ссылки на объект, на экземпляр Cafe. Теперь назначим cafe и newCafe значение nil.

```
cafe = nil
newCafe = nil
```

Обратите внимание, что в консоль все еще выводится только слово “Инициализация”.

```
127  cafe = nil
128  newCafe = nil
```

Инициализация

Это потому, что объект еще не освобожден, на него все еще осталась одна сильная ссылка. То есть когда мы присвоили cafe значение nil, количество сильных ссылок уменьшилось на одну, стало равно двум. Затем мы присвоили newCafe значение nil и количество ссылок уменьшилось еще на один. Таким образом количество ссылок на объект сейчас равно 1 и он все еще существует в памяти.

Теперь присвоим superNewCafe значение nil.

```
superNewCafe = nil
```

Запустим код и увидим, что теперь в консоли появилось “Деинициализация”, а значит ARC освободил экземпляр Cafe, это произошло, так как теперь количество ссылок равно 0.

```
127  cafe = nil
128  newCafe = nil
129  superNewCafe = nil
```

Инициализация
Деинициализация

Ссылки

Ранее упоминалось словосочетание “сильные ссылки”, А что это за ссылки? Раз есть сильные, значит есть и слабые? Да, в Swift есть несколько видов ссылок.

1. strong - сильная ссылка, пока на объект есть сильная ссылка, он не будет удален из памяти.
2. weak - слабая ссылка, является указателем на объект, но не увеличивает счетчик ссылок, поэтому объект, на который есть только weak ссылка бужет удален из памяти. Слабая ссылка является опционалом, поэтому данная ссылка будет изменяться на nil, если на объект больше не указывает ни одной сильной ссылки. Чтобы создать слабую ссылку необходимо указать weak перед объектом, например, создадим слабую ссылку на объект Cafe

```
weak var cafe: Cafe?
```

🔥 weak ссылки обязательно опционалы, поэтому после Cafe стоит вопросительный знак

3. unowned - бесхозные ссылки, они схожи со слабыми ссылками, то есть количество ссылок на объект не увеличивается. Однако unowned ссылка не является опционаллом. Данную ссылку рекомендуется использовать, если вы уверены, что объект никогда не станет nil, в противном случае пройзойдет краш приложения. Чтобы создать бесхозную ссылку на объект необходимо указать unowned.

```
unowned var cafe: Cafe
```

Главная причина по которой мы должны использовать слабые и бесхозные ссылки - цикл сильных ссылок. Это ситуация, когда объекты удерживают друг друга сильными ссылками и память не может быть очищена. Как пример, допустим у нас есть классы Cake и Buyer. В классе Cake есть имя, стоимость и покупатель, который купил этот десерт. В классе Buyer есть имя покупателя и десерт, который он купил.

```
class Cake {
    var name: String = "Cake"
    var cost: Double = 100
    var buyer: Buyer?
}

class Buyer {
    var name: String = "Tom"
    var cake: Cake?
}
```

Теперь добавим в каждый класс деинициализатор, чтобы отследить, когда память будет очищена.

```
class Cake {
    var name: String = "Cake"
    var cost: Double = 100
    var buyer: Buyer?

    deinit {
        print("Деинициализация")
    }
}

class Buyer {
    var name: String = "Tom"
    var cake: Cake?

    deinit {
        print("Деинициализация")
    }
}
```

Теперь создадим экземпляры классов

```
var cake: Cake? = Cake()
var buyer: Buyer? = Buyer()
```

А теперь назначим эти объекты друг другу

```
cake?.buyer = buyer
buyer?.cake = cake
```

Объекты начали ссылаться друг на друга сильными ссылками. Попробуем очистить их. Присвоим обеим переменным nil

```
92 cake = nil
93 buyer = nil
```

Как мы видим, в консоль не было выведено ни одной “Деинициализации”, а ведь должно быть сразу две. Это произошло потому, что мы создали цикл сильных ссылок, Cake сильно держит Buyer, а Buyer сильно держит Cake, поэтому счетчик ссылок не станет равен 0, а значит память не будет очищена и выходит что у нас появляется утечка памяти. Избежать таких ситуаций и помогают сильные и бесхозные ссылки. Давайте подумаем, покупатель может выйти из кафе без десерта? Да. А десерт без покупателя? Нет. Поэтому есть смысл, чтобы покупатель держал десерт слабой ссылкой, ведь он вполне может уйти без него. А вот десерту

необходимо держать покупателя сильной ссылкой, ведь без него десерту проблематично покинуть заведение.

```
class Cake {
    var name: String = "Cake"
    var cost: Double = 100
    var buyer: Buyer?

    deinit {
        print("Деинициализация")
    }
}

class Buyer {
    var name: String = "Tom"
    weak var cake: Cake?

    deinit {
        print("Деинициализация")
    }
}
```

Теперь еще раз попробуем обеим переменным присвоить значение nil.

```
91
92 cake = nil
93 buyer = nil
```

Деинициализация
Деинициализация

Деинициализатор был вызван, а значит память была очищена и мы избежали цикла сильных ссылок!

Захват значений в замыканиях

У замыканий в Swift существует возможность захвата значений, то есть они могут сохранять начальные значения переданных в них переменных. Например есть две переменных: a и b, которые равны 5 и 10 соответственно. А теперь рассмотрим замыкание, которое возвращает результат перемножения этих двух переменных.

```
var multiply: () -> Int = {
    a * b
}
```

И теперь выведем результат перемножения

```
260 print(multiply())
```

```
50
```

Теперь изменим значения a и b, и еще раз вызовем функцию

```
262 a = 3
263 b = 4
264 print(multiply())
```

```
12
```

Значение изменилось, однако у нас есть возможность “захватить” начальные переменные. Для этого в квадратных скобках укажем переменные, которые хотим захватить, это a и b

```
var multiply: () -> Int = { [a, b] in
    a * b
}
```

Проверим, какое значение будет выведено в консоль

```
260 print(multiply())
```

```
50
```

А теперь изменим значения a и b на 3 и 4 соответственно.

```
262 a = 3
263 b = 4
264 print(multiply())
```

```
50
```

Значение все еще будет 50, так как переменные a и b в квадратных скобках были захвачены, то есть мы зафиксировали их начальные значения, и даже если затем эти переменные переопределить - замыкание будет оперировать прежним значением.

Также стоит учитывать, что если у нас есть замыкание внутри класса и внутри замыкания вызываются свойства и переменные этого класса - класс будет по умолчанию удерживать сильной ссылкой. Что имеется в виду: например, у нас есть класс TestClass, в котором есть две переменные, а также два метода: один принимает на вход замыкание, а второй метод вызывает первый.

```
class TestClass {
    private var a = 5
    private var b = 7
```

```
    func start(handler: @escaping () -> Void) {
        handler()
    }

    func call() {
        start {
            let c = a + b
            print(c)
        }
    }
}
```

В данный момент этот код не запустится, будет ошибка, что перед a и b необходимо указать self

```
    func call() {
        start {
            let c = a + b
            print(c)
        }
    }
```

Reference to property 'a' in closure requires explicit use of 'self' to make capture semantics explicit

Давайте добавим self перед a и b

```
func call() {
        start {
            let c = self.a + self.b
            print(c)
        }
    }
```

Теперь, когда мы указали self, мы по умолчанию захватили класс TestClass и теперь на него есть еще одна сильная ссылка. Нужно быть осторожней с такими ситуациями, чтобы не получился цикл сильных ссылок, когда класс держит замыкание, а замыкание держит класс, Чтобы этого избежать нужно указать, что мы захватываем класс слабой или бесхозной ссылкой. Для этого в квадратных скобках необходимо указать weak или unowned перед self. Давайте будем держать класс слабой ссылкой.

```
start { [weak self] in
            let c = self.a + self.b
            print(c)
        }
```

Обратите внимание, что теперь self внутри замыкания является опционалом и можно, например, распаковать его при помощи guard let

```
func call() {
        start { [weak self] in
            guard let self = self else {
```

```
                return
            }
            let c = self.a + self.b
            print(c)
        }
    }
```

Copy-on-write в коллекциях

Коллекции в Swift являются типом значения, однако для них реализован механизм copy-on-write. Что же это такое?

Copy-on-write - механизм, который копирует поведение reference type для value type. Это значит, что до первых изменений объекты, в которых хранятся одинаковые коллекции, будут ссылаться на одну и ту же область.

Например, создадим массив строк.

```
var animals = ["Cat", "Dog", "Cow"]
```

Теперь создадим новую переменную и в нее присвоим значение animals.

```
var newAnimals = animals
```

И напишем функцию, которая будет печатать в консоль адрес в памяти.

```
func getAddress(_ collection: UnsafeRawPointer) {
    print(Int(bitPattern: collection))
}
```

Теперь вызовем эту функцию и для animals, и для newAnimals

```
246 getAddress(animals)
247 getAddress(newAnimals)
```

```
105553176686976
105553176686976
```

Как мы видим, обе переменные, хоть и являются типом значения, ссылаются на одну область памяти. Теперь изменим newAnimals.

```
245
246 newAnimals.append("Tiger")
247
248 getAddress(animals)
249 getAddress(newAnimals)
```

```
105553180682832
105553167004576
```

Теперь переменные ссылаются на разные области памяти. Таким образом, механизм copy-on-write избавляет программу от ненужных копирований и повышает производительность наших приложений.

Приведение типов

Теперь мы перейдем к заключительной теме этого курса - приведение типов. Приведение типов используется, когда необходимо использовать экземпляр одного класса, как часть другого класса в той же иерархии классов. При использовании приведения типов мы обращаемся к объекту одного типа, как к объекту другого типа.

Например, сначала у нас есть класс Shop, в котором продаются только продукты

```
class Shop {
    var products: [Product]

    init(products: [Product]) {
        self.products = products
    }
}
```

Затем мы добавили класс MiniMarket, в котором продается еще и бытовая химия. Класс является наследником Shop

```
class MiniMarket: Shop {
    var householdChemicals: [HouseholdChemicals]

    init(products: [Product], householdChemicals:
[HouseholdChemicals]) {
        self.householdChemicals = householdChemicals
        super.init(products: products)
    }
}
```

Далее добавим класс GiperMarket, в котором продается еще и одежда. Класс является наследником MiniMarket

```
class GiperMarket: MiniMarket {
```

```
    var clothes: [Clothes]

    init(products: [Product], householdChemicals:
[HouseholdChemicals], clothes: [Clothes]) {
        self.clothes = clothes
        super.init(products: products, householdChemicals:
householdChemicals)
    }
}
```

И теперь мы можем создать переменную с типом Shop, но значение будет типа GiperMarket и даже когда мы вызовем функцию type(of:) тип будет GiperMarket

```
739 let giperMarket: Shop = GiperMarket(products: [], householdChemicals:
        [], clothes: [])
740 print(type(of: giperMarket))
```

```
GiperMarket
```

Но, так как изначально мы обозначили переменную типа Shop получить свойство clothes мы не сможем.

```
740 giperMarket.clothes        Value of type 'Shop' has no member 'clothes'
```

Справиться с этой ситуацией нам помогут ключевые слова is и as.

is

При помощи ключевого слова is мы можем проверить тип. Выражение с is возвращает переменную с типом Bool.

```
743
744 print(giperMarket is Shop)
745 print(giperMarket is MiniMarket)
746 print(giperMarket is GiperMarket)
747 print(giperMarket is Cafe)
```

```
true
true
true
false
```

Когда мы используем is вместе с giperMarket и Shop в консоль будет напечатано true, так как по сути giperMarket является типом Shop, ведь он его наследник.
Когда мы используем is вместе с giperMarket и MiniMarket в консоль будет напечатано true, так как по сути giperMarket является типом MiniMarket, ведь он его наследник.

Когда мы используем is вместе с giperMarket и GiperMarket в консоль будет напечатано true, так как по сути giperMarket является типом GiperMarket.
Когда мы используем is вместе с giperMarket и Cafe в консоль будет напечатано false, так как по сути giperMarket никак не связан с Cafe.
Обратите внимание, что, если использовать is между miniMarket и GiperMarket будет возвращено false, так как MiniMarket не является наследником GiperMarket

```
749 let miniMarket = MiniMarket(products: [], householdChemicals: [])
750 print(miniMarket is GiperMarket)
```

```
false
```

Принудительное преобразование

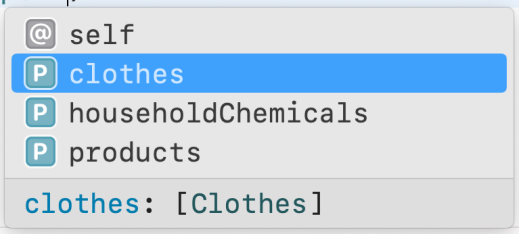
Для принудительного преобразования используется as!, выражение с as! вернет принудительно извлеченное значение. Использовать можно только тогда, когда есть уверенность, что преобразование будет успешно, иначе приложение упадет.

```
744 print(giperMarket as! Shop)
745 print(giperMarket as! MiniMarket)
746 print(giperMarket as! GiperMarket)
```

```
__lldb_expr_27.GiperMarket
__lldb_expr_27.GiperMarket
__lldb_expr_27.GiperMarket
```

Также теперь мы можем положить значение в переменную и получить параметр clothes, хоть и указали изначально тип Shop

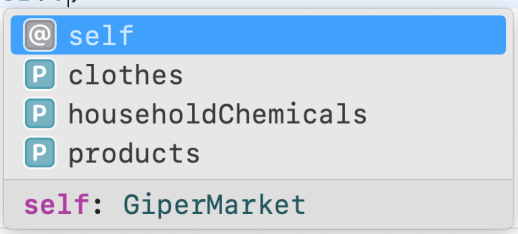
```
748 let giper = giperMarket as! GiperMarket
749 print(giper.)
750
751
752
753
754
755
```



Опциональное преобразование

Так как принудительное преобразование достаточно опасно из-за возможной фатальной ошибки в случаи неудачи, зачастую выгоднее использовать опциональное преобразование при помощи as?. Работает оно точно также, как и принудительное, но возвращает опциональное значение.

```
748 let giper = giperMarket as? GiperMarket
749 print(giper?.)
750
751
752
753
754
755
```



А в случае неудачи будет просто возвращен nil

```
748 let giper = giperMarket as? Cafe
749 print(giper)
750
```

nil

Также as можно использовать и с протоколами, например, у нас есть две структуры: Cake и Eclair

```
struct Eclair {
    var taste: String
    var cost: Double
}

struct Cake {
    var taste: String
    var cost: Double
}
```

И есть протокол, которому они соответствуют

```
protocol Dessert {
    var taste: String { get set }
    var cost: Double { get set }
}

struct Eclair: Dessert {
    var taste: String
    var cost: Double
}

struct Cake: Dessert {
    var taste: String
    var cost: Double
}
```

Теперь создадим массив, в котором будут хранится элементы, соответствующие протоколу Dessert. Положим в него несколько элементов Eclair и Cake.

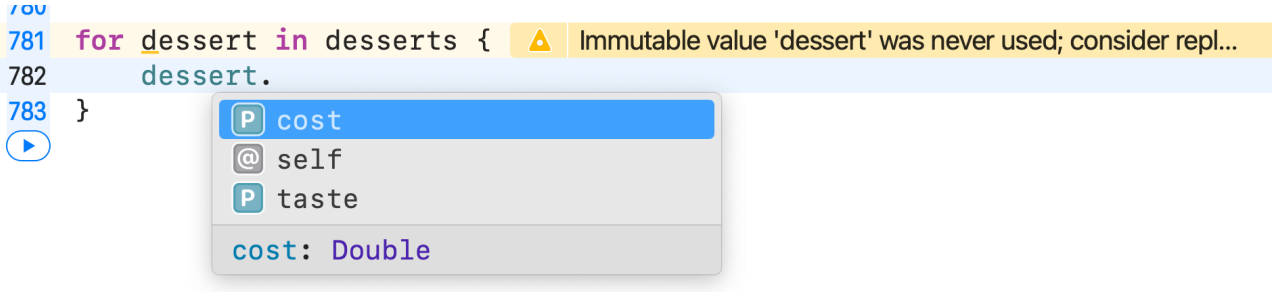
```
var desserts: [Dessert] = [Cake(taste: "a", cost: 110),
```

```
Eclair(taste: "b", cost: 200), Cake(taste: "c", cost: 150),
Cake(taste: "d", cost: 125)]
```

Далее укажем в структуру Cake переменную color.

```
struct Cake: Dessert {
    var taste: String
    var cost: Double
    var color: String = "default"
}
```

И напишем функцию, выведем в консоль для всех элементов Cake. И здесь у нас начинается проблема, ведь массив desserts хранит в себе элементы Dessert, а не Cake. А в dessert не указано свойство color



Здесь на помощь придет as. Преобразуем элемент из Dessert в Cake, а если неудачно, то не будем ничего с ним делать.

```
781 for dessert in desserts {
782     if let dessert = dessert as? Cake {
783         print(dessert.color)
784     }
785 }
```

```
default
default
default
```

Что можно почитать еще?

1. SwiftBook. Автоматический подсчет ссылок
2. SwiftBook. Приведение типов

Используемая литература

1. https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html
2. https://docs.swift.org/swift-book/LanguageGuide/Closures.html
3. https://docs.swift.org/swift-book/LanguageGuide/TypeCasting.html