






# ARC и управление памятью

Урок 6





## Что будет на уроке сегодня

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



# Reference и value type



## Reference 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`



## Копирование значений

```
1 class Cafe: CafeProtocol {  
2     var coffee: [Coffee]  
3     var tea: [Tea]  
4     var name: String  
5  
6     // код  
7 }
```



## Копирование значений

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

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





## Копирование значений

И `safe`, и `newSafe` теперь указывают на один и тот же объект в памяти, поэтому, если мы изменим название в `newSafe`, то название `safe` тоже изменится. Как мы видим, после того, как мы изменили название `newSafe`, название `safe` тоже изменилось на "New safe"

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



**Название safe New safe**  
**Название newSafe New safe**



## Копирование значений

Теперь же рассмотрим структуру Coffee

```
1 struct Coffee {  
2     enum CoffeeSize {  
3         case s  
4         case m  
5         case l  
6     }  
7  
8     var name: String  
9     var isSugar: Bool  
10    var isIce: Bool  
11    var cost: Double = 110  
12    var size: CoffeeSize  
13 }
```



## Копирование значений

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

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



## Копирование значений

Структура имеет тип значения, поэтому, когда мы в переменную newLatte кладем значение переменной latte, происходит копирование значения, 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: `safe` и `latte`. `safe` это класс, а `coffee` - структура, попробуем изменить название и кафе и кофе

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



## Изменение константы

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.

```
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



## Оператор идентичности

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

```
1 var safe = Cafe(name: "Cafe")  
2 var newCafe = safe
```

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

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



## Оператор идентичности

Начнем сравнивать экземпляры объектов. В данном случае будет выведено true, так как объекты ссылаются на одну и ту же область памяти

108

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



**true**





## Оператор идентичности

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**



## Оператор идентичности

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



## 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.

```
1 class Cafe: CafeProtocol {
2     var coffee: [Coffee]
3     var tea: [Tea]
4     var name: String
5
6     required init(coffee: [Coffee], tea: [Tea], name: String) {
7         self.coffee = coffee
8         self.tea = tea
9         self.name = name
10
11         print("Инициализация")
12     }
13     // код
14 }
```



## Как работает ARC

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

```
1 deinit {  
2     print("Деинициализация")  
3 }
```



## Как работает ARC

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

```
1 var cafe: Cafe?  
2 var newCafe: Cafe?  
3 var superNewCafe: Cafe?
```



## Как работает ARC

Теперь создадим новый экземпляр и присвоим его одной из переменных

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





## Как работает ARC

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**



## Как работает ARC

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

```
1 cafe = Cafe(name: "Cafe")  
2 newCafe = cafe  
3 superNewCafe = cafe
```



## Как работает ARC

Теперь назначим `safe` и `newCafe` значение `nil`.

```
1 cafe = nil
2 newCafe = nil
```



## Как работает ARC

Обратите внимание, что в консоль все еще выводится только слово “Инициализация”. Это потому, что объект еще не освобожден, на него все еще осталась одна сильная ссылка. То есть когда мы присвоили `safe` значение `nil`, количество сильных ссылок уменьшилось на одну, стало равно двум. Затем мы присвоили `newSafe` 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.

```
127  safe = nil
128  newSafe = nil
```



**Инициализация**



## Как работает ARC

Теперь присвоим superNewCafe значение nil.

```
1 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
130
```



**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 будет 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 ссылки обязательно опционалы, поэтому после Cafe стоит вопросительный знак

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

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

```
1 unowned var cafe: Cafe
```



## Ссылки

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.

```
1 class Cake {  
2     var name: String = "Cake"  
3     var cost: Double = 100  
4     var buyer: Buyer?  
5 }  
6  
7 class Buyer {  
8     var name: String = "Tom"  
9     var cake: Cake?  
10 }
```



## Ссылки

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

```
1 class Cake {
2     var name: String = "Cake"
3     var cost: Double = 100
4     var buyer: Buyer?
5
6     deinit {
7         print("Деинициализация")
8     }
9 }
10
11 class Buyer {
12     var name: String = "Tom"
13     var cake: Cake?
14
15     deinit {
16         print("Деинициализация")
17     }
18 }
```





## Ссылки

Теперь создадим экземпляры классов

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

А теперь назначим эти объекты друг другу

```
1 cake?.buyer = buyer  
2 buyer?.cake = cake
```



## Ссылки

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

```
91  
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.



## Ссылки

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.

```
1 class Cake {
2     var name: String = "Cake"
3     var cost: Double = 100
4     var buyer: Buyer?
5
6     deinit {
7         print("Деинициализация")
8     }
9 }
10
11 class Buyer {
12     var name: String = "Tom"
13     weak var cake: Cake?
14
15     deinit {
16         print("Деинициализация")
17     }
18 }
```



## Ссылки

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



## 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.

```
1 var multiply: () → Int = {  
2     a * b  
3 }
```



## Захват значений в замыканиях

И теперь выведем результат перемножения

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



**50**

Теперь изменим значения `a` и `b`, и еще раз вызовем функцию. Значение изменилось, однако у нас есть возможность “захватить” начальные переменные

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



**12**



## Захват значений в замыканиях

Для “захвата” начальных значений в квадратных скобках укажем переменные, которые хотим захватить, это a и b

```
1 var multiply: () → Int = { [a, b] in  
2   a * b  
3 }
```





## Захват значений в замыканиях

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.



## Захват значений в замыканиях

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.

```
1 class TestClass {
2     private var a = 5
3     private var b = 7
4     func start(handler: @escaping () -> Void) {
5         handler()
6     }
7
8     func call() {
9         start {
10             let c = a + b
11             print(c)
12         }
13     }
14 }
```



## Захват значений в замыканиях

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

```
1 func call() {  
2     start {  
3         let c = self.a + self.b  
4         print(c)  
5     }  
6 }
```



## Захват значений в замыканиях

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.

```
1 start { [weak self] in
2         let c = self.a + self.b
3         print(c)
4     }
```



## Захват значений в замыканиях

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

```
1 func call() {  
2     start { [weak self] in  
3         guard let self = self else {  
4             return  
5         }  
6         let c = self.a + self.b  
7         print(c)  
8     }  
9 }
```



# Copy-on-write в коллекциях



## 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.



## Copy-on-write в коллекциях

Например, создадим массив строк.

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

Теперь создадим новую переменную и в нее присвоим значение animals.

```
1 var newAnimals = animals
```

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

```
1 func getAddress(_ collection: UnsafeRawPointer) {  
2     print(Int(bitPattern: collection))  
3 }
```





## Copy-on-write в коллекциях

Теперь вызовем эту функцию и для `animals`, и для `newAnimals`. Как мы видим, обе переменные, хоть и являются типом значения, ссылаются на одну область памяти.

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



**105553176686976**

**105553176686976**



## Copy-on-write в коллекциях

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



## 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

```
1 class Shop {  
2     var products: [Product]  
3  
4     init(products: [Product]) {  
5         self.products = products  
6     }  
7 }
```



## Приведение типов

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

```
1 class MiniMarket: Shop {
2     var householdChemicals: [HouseholdChemicals]
3
4     init(products: [Product], householdChemicals: [HouseholdChemicals]) {
5         self.householdChemicals = householdChemicals
6         super.init(products: products)
7     }
8 }
```



## Приведение типов

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

```
1 class GiperMarket: MiniMarket {
2     var clothes: [Clothes]
3
4     init(products: [Product], householdChemicals: [HouseholdChemicals],
5         clothes: [Clothes]) {
6         self.clothes = clothes
7         super.init(products: products, householdChemicals:
8             householdChemicals)
9     }
10 }
```



## Приведение типов

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`. 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`.

```
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
```





## is

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)|
```



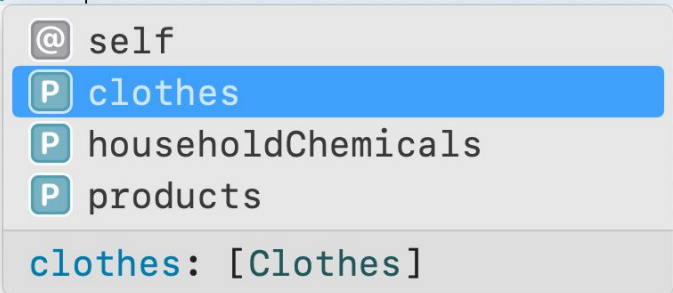
```
__lldb_expr_27.GiperMarket
__lldb_expr_27.GiperMarket
__lldb_expr_27.GiperMarket
```



## Принудительное преобразование

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
```



- @ self
- P clothes**
- P householdChemicals
- P products

clothes: [Clothes]





## Опциональное преобразование

Так как принудительное преобразование достаточно опасно из-за возможной фатальной ошибки в случае 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
```



self



clothes



householdChemicals



products

self: GiperMarket





## Опциональное преобразование

А в случае неудачи будет просто возвращен nil

```
748 let giper = giperMarket as? Cafe
```

```
749 print(giper)|
```

```
750
```



**nil**



## Опциональное преобразование

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

```
1 struct Eclair {  
2     var taste: String  
3     var cost: Double  
4 }  
5  
6 struct Cake {  
7     var taste: String  
8     var cost: Double  
9 }
```



## Опциональное преобразование

И есть протокол, которому они соответствуют

```
1 protocol Dessert {  
2     var taste: String { get set }  
3     var cost: Double { get set }  
4 }  
5  
6 struct Eclair: Dessert {  
7     var taste: String  
8     var cost: Double  
9 }  
10  
11 struct Cake: Dessert {  
12     var taste: String  
13     var cost: Double  
14 }
```



## Опциональное преобразование

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

```
1 var desserts: [Dessert] = [Cake(taste: "a", cost: 110), Eclair(taste: "b",  
    cost: 200), Cake(taste: "c", cost: 150), Cake(taste: "d", cost: 125)]
```





## Опциональное преобразование

Далее укажем в структуру Cake переменную color.

```
1 struct Cake: Dessert {  
2     var taste: String  
3     var cost: Double  
4     var color: String = "default"  
5 }
```



## Опциональное преобразование

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`

```
781 for dessert in desserts {  
782     dessert.  
783 }
```



Immutable value 'dessert' was never used; consider repl...

 `cost`

 `self`

 `taste`

`cost: Double`



## Опциональное преобразование

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**



# Заключение