Урок 4

# Введение в ООП

Как описывать объекты реального мира и создавать их по описанию. Классы и объекты. Наследование. ARC и введение в управление памятью.

# Классы

На прошлом уроке мы познакомились со структурами. Так как в Swift структуры очень похожи на классы, вы уже почти знаете и классы. Но отличия всё же есть: структурам недоступны некоторые ключевые особенности классов.

## Ссылочный тип

Возьмем структуру «Honda», которую мы написали на прошлом уроке, и немного поэкспериментируем с машинами. Создадим две разных машины с разными параметрами пробега. Выведем в консоль их пробег. После присвоим первой переменной значение второй и снова выведем в консоль их пробег. Последний шаг – изменим пробег первой машины и в третий раз выведем пробег в консоль.

```
enum HondaDoorState {
    case open, close
}
enum Transmission {
    case manual, auto
}
struct Honda {
    let color: UIColor
    let mp3: Bool
    let transmission: Transmission
    var km: Double
    var doorState: HondaDoorState
}
var car1 = Honda(color: .white, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)
var car2 = Honda(color: .white, mp3: false, transmission: .auto, km: 120.0,
doorState: .close)
print(car1.km,", ", car2.km)    //0.0, 120.0
car2 = car1
print(car1.km,", ", car2.km)  //0.0, 0.0
car1.km = 500.0
print(car1.km,", ", car2.km) //500.0, 0.0
```

В первый раз в консоль вывелись разные цифры, ведь это были разные машины с разным пробегом. После того как мы провели присваивание, цифры в консоли стали одинаковыми, потому что теперь там одинаковые машины. И наконец, после изменения пробега у первой машины цифры в консоли снова стали отличаться, ведь у второй машины мы пробег не меняли.

Такое поведение и ожидает увидеть большинство новичков, в этом нет ничего удивительного. Но давайте всё же разберем, как это произошло. Структуры, как и простые типы данных, являются типом значения. Это означает, что в переменных хранится само значение структуры. Когда мы присваиваем значение одной переменной другой, мы фактически полностью копируем содержимое переменной. После создания копии в обеих переменных находятся два разных экземпляра структуры, но с одинаковыми значениями. Ближайшая аналогия – ксерокс: если скопировать документ, у вас будет два разных листа с одинаковым содержимым. Если вы напишете что-нибудь на одном листе, второй не изменится.

Теперь давайте превратим нашу структуру в класс. Для этого измените ключевое слово «struct» на «class». И вы увидите ошибку. Мы только что столкнулись с первым отличием класса и структуры.

Дело в том, что класс не предоставляет конструктора по умолчанию для инициализации всех свойств. Добавьте конструктор, как мы делали это на прошлом уроке.

```
enum HondaDoorState {
    case open, close
}
enum Transmission {
    case manual, auto
}
class Honda {
    let color: UIColor
    let mp3: Bool
    let transmission: Transmission
    var km: Double
    var doorState: HondaDoorState
    init(color: UIColor, mp3: Bool, transmission: Transmission, km: Double,
doorState: HondaDoorState) {
        self.color = color
        self.mp3 = mp3
        self.transmission = transmission
        self.km = km
        self.doorState = doorState
    }
}
var car1 = Honda(color: .white, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)
var car2 = Honda(color: .white, mp3: false, transmission: .auto, km: 120.0,
doorState: .close)
print(car1.km,", ", car2.km)//0.0, 120.0
car2 = car1
print(car1.km,", ", car2.km)//0.0, 0.0
car1.km = 500.0
print(car1.km,", ", car2.km)//500.0, 500.0
```

Давайте посмотрим на вывод в консоль после манипуляций. Первый и второй вывод остались без изменений, а вот в третий раз пробеги одинаковые. Мы изменили пробег только у одной машины – как вышло, что изменились обе?

Причина кроется в том, что класс – это ссылочный тип. Это означает, что в переменной хранится не само значение, а ссылка на него. Когда мы присваиваем одной переменной типа класс значение другой переменной, мы копируем не сам объект, а ссылку. В результате у нас две переменных, которые содержат ссылки на один и тот же объект в памяти. В качестве примера можно упомянуть видео на YouTube. Если вы выложите на свой канал видео и отправите другу ссылку на него, вы не копируете само видео. Если вы после этого измените ролик, например, добавите титры, то и друг по ссылке откроет это же видео, но уже с титрами.

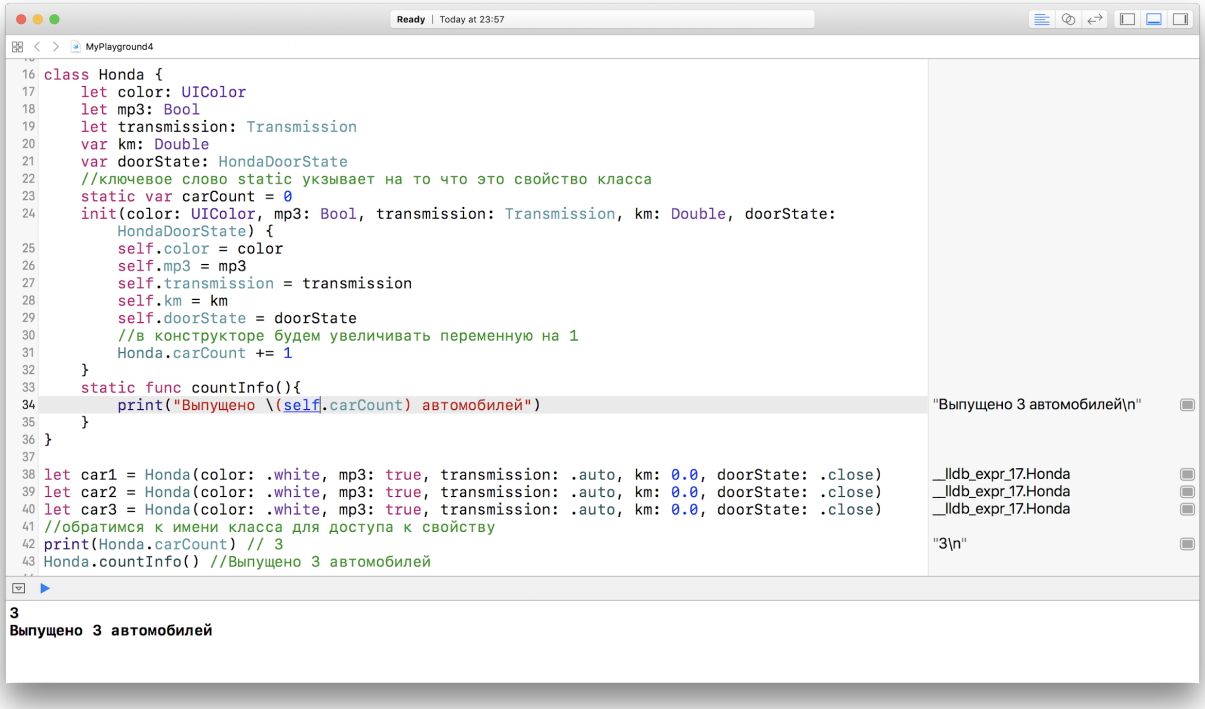
## Свойства и методы класса

Мы уже достаточно понимаем концепцию «класс-объект», чтобы поговорить о свойствах и методах класса. Все свойства и методы, что мы объявляли, принадлежали объектам. Это означает, что только готовый автомобиль имеет свойства пробега, трансмиссии и прочие. Только у готового автомобиля имеются методы открытия и закрытия дверей. Это логично – чертеж не может открывать и закрывать двери. Но дело в том, что у класса могут быть свои свойства. Они доступны только при обращении к

классу. Он существует в единственном экземпляре, поэтому когда бы мы к нему ни обратились, мы будем читать и писать одно и то же свойство. Это же касается и методов. Вы даже имели дело с особенным методом класса – это конструктор. Если помните, мы вызывали его, обращаясь к классу, а вот у инициализированной машины конструктор уже недоступен.

Представьте, что нам необходимо следить за количеством выпущенных машин. Для этого идеально подойдет свойство класса. Кроме того, добавим метод класса для вывода этого количества в консоль.

```
class Honda {
    let color: UIColor
    let mp3: Bool
    let transmission: Transmission
    var km: Double
    var doorState: HondaDoorState
// ключевое слово static указывает на то, что это свойство класса
    static var carCount = 0
    init(color: UIColor, mp3: Bool, transmission: Transmission, km: Double,
doorState: HondaDoorState) {
        self.color = color
        self.mp3 = mp3
        self.transmission = transmission
        self.km = km
        self.doorState = doorState
// в конструкторе будем увеличивать переменную на 1
        Honda.carCount += 1
    }
    static func countInfo(){
        print("Выпущено \(self.carCount) автомобилей")
    }
}
let car1 = Honda(color: .white, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)
let car2 = Honda(color: .white, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)
let car3 = Honda(color: .white, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)
// обратимся к имени класса для доступа к свойству
print(Honda.carCount)     // 3
Honda.countInfo()         // Выпущено 3 автомобилей
}
let car1 = Honda(color: .white, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)
let car2 = Honda(color: .white, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)
let car3 = Honda(color: .white, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)
//обратимся к имени класса для доступа к свойству
print(Honda.carCount)    // 3
```




Важно понимать, что свойства и методы класса не связаны с его объектами. Вы не сможете обратиться к ним из методов объекта по ссылке «self», зато сможете из метода класса. Фактически в объекте «self» указывает на объект, а в классе – на класс.

Вы можете обратиться к свойствам и методам класса из любого места, в том числе из самого объекта, используя для этого имя класса «Honda.carCount». Но вот из класса к объекту вы обратиться не сможете.

Структуры и перечисления тоже могут иметь свойства и методы классов.

## Вспомогательные конструкторы

На прошлом уроке мы узнали, что такое инициализаторы. Напомню, это обязательные методы класса, необходимые для создания экземпляров класса или структуры. Классы тоже имеют инициализаторы, точно такие же, как в структурах. Но кроме простых инициализаторов имеются еще и вспомогательные. Обычные инициализаторы являются обязательными, т.е. без них нельзя создать объект. Без вспомогательных можно обойтись, они нужны для более удобного создания объектов в определенных случаях. Вспомогательный инициализатор обязан вызвать обычный инициализатор.

Например, у нас есть класс прямоугольника, у него есть два обязательных свойства – одна сторона и другая. Мы не можем создать объект, не указав длины сторон, поэтому мы обязаны определить инициализатор для этого. Но если нам часто приходится создавать квадратные прямоугольники, мы можем добавить вспомогательный конструктор для этой цели.

Желательно делать все необязательные конструкторы вспомогательными.

```
class Rectangle {
    var sideA: Double                       // обязательные переменные
    var sideB: Double
    init(sideA: Double, sideB: Double) { // обязательный конструктор
        self.sideA = sideA
        self.sideB = sideB
    }
    convenience init(side: Double){     // вспомогательный  конструктор
        self.init(sideA: side, sideB: side)
    }
}
```

# Наследование

Допустим, пока мы с вами писали этот код, руководство фирмы решило, что не стоит ограничиваться одной моделью машины, и поручило инженерам разработать вторую. «Honda Sport» должна быть точной копией обычной «Honda», но у неё должен быть люк и механизмы его открытия/закрытия.

Как поступить в данной ситуации? Можно создать ещё один класс с необходимым функционалом, но мы сделаем проще: скопируем код определения нашей «Honda» и допишем недостающее.

```
class HondaSport {
    let color: UIColor
    let mp3: Bool
    let transmission: Transmission
    var km: Double
    var doorState: HondaDoorState
    var hatchState: HondaHatchState // состояние люка
    // Также изменим конструктор
    init(color: UIColor, mp3: Bool, transmission: Transmission, km: Double,
doorState: HondaDoorState, hatchState: HondaHatchState) {
        self.color = color
        self.mp3 = mp3
        self.transmission = transmission
        self.km = km
        self.doorState = doorState
        self.hatchState = hatchState
    }
    // нам надо открывать люк
    func openHatch() {
        hatchState = .open
    }
   // и закрывать
    func closeHatch() {
        hatchState = .close
    }
}
var car1 = Honda(color: .white, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)
var sportCar1 = HondaSport(color: .red, mp3: true, transmission: .manual, km:
0.0, doorState: .close, hatchState: .close)
```

Теперь мы можем создавать спортивные автомобили. Но представьте, что таких похожих моделей нам надо 20 штук. Не хотелось бы копировать и править их 20 раз подряд, но и это не самое страшное. Представьте, что наша фирма решит поставить во все модели холодильник – нам придется 20 раз внести одинаковые правки во все классы. Это очень скучная и неблагодарная работа. Именно так рассуждали авторы принципов ООП, когда подарили нам наследование.

**Наследование** – это возможность создавать новые классы потомками уже имеющихся. По умолчанию потомок полностью копирует родителя, но ему можно добавить свои классы и методы. Давайте изменим спорткар, объявим его через наследование.

```
class Honda {
    let color: UIColor
    let mp3: Bool
    let transmission: Transmission
    var km: Double
    var doorState: HondaDoorState
    init(color: UIColor, mp3: Bool, transmission: Transmission, km: Double,
doorState: HondaDoorState) {
        self.color = color
        self.mp3 = mp3
        self.transmission = transmission
        self.km = km
        self.doorState = doorState
    }
}
class HondaSport: Honda { // наследуем HondaSport от Honda
// мы ничего не пишем здесь
// и наш новый класс имеет все те же свойства и методы, что и его родитель
}
var car1 = Honda(color: .white, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)
// мы можем создать объект спорткара
var sportCar1 = HondaSport(color: .white, mp3: true, transmission: .auto, km:
0.0, doorState: .close)
```

Теперь осталось только добавить все необходимые свойства и методы. Правда, есть один нюанс. Как только вы добавите новое свойство, требующее инициализации, вы получите ошибку. Конструктор родительского класса ничего не знает о нем, и надо написать новый конструктор. Список параметров должен быть полным, включая все параметры, полученные от родителя. А вот присваивание свойств в его реализации будет неполным. Мы обязаны воспользоваться родительским конструктором, чтобы завершить создание объекта. Обратите внимание, что новые свойства мы инициализируем до того, как воспользуемся родительским конструктором. Родительский конструктор вызывается через ссылку «super». Она похожа на «self» и также присутствует во всех методах классов, но ссылается она не на этот объект, а на методы и свойства родителя.

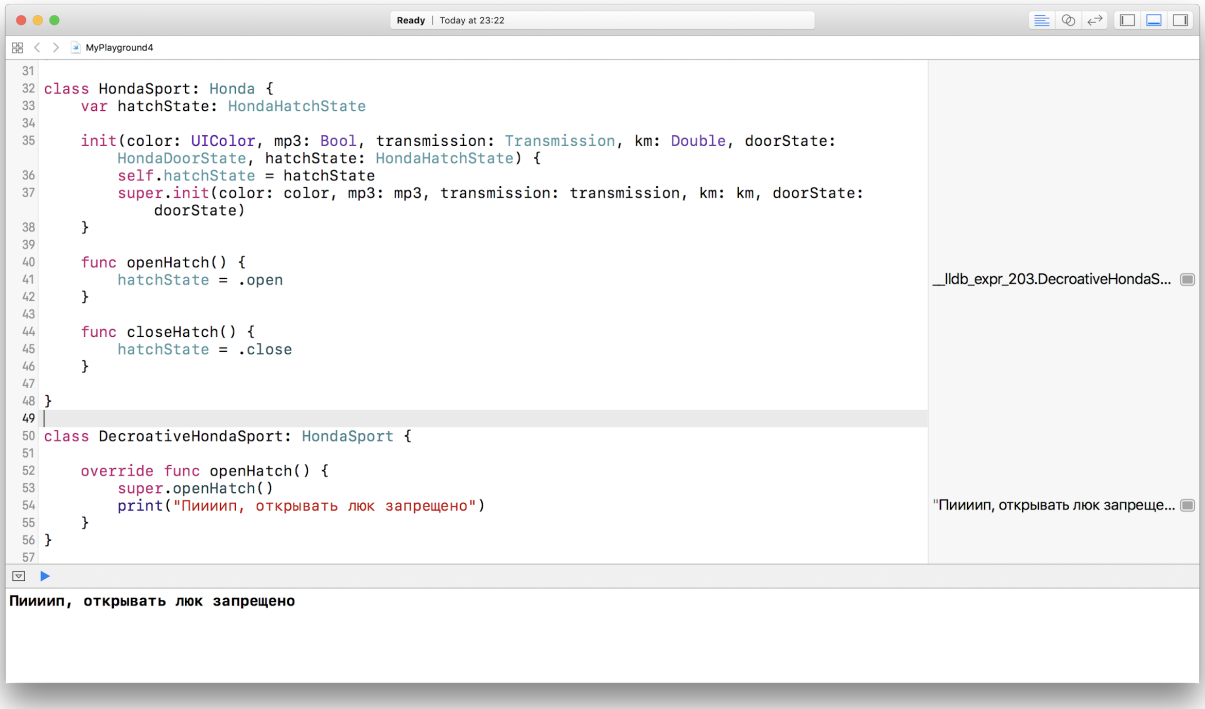
```
class HondaSport: Honda {
    var hatchState: HondaHatchState      // Новое свойство
    // Перечисляем все свойства
    init(color: UIColor, mp3: Bool, transmission: Transmission, km: Double,
doorState: HondaDoorState, hatchState: HondaHatchState) {
        self.hatchState = hatchState    // инициализируем новое свойство
```

```
    // используем конструктор из родителя, чтобы завершить инициализацию
класса
        super.init(color: color, mp3: mp3, transmission: transmission, km: km,
doorState: doorState)
    }

    func openHatch() {                  // Новый метод
        hatchState = .open
    }

    func closeHatch() {                 // Новый метод
        hatchState = .close
    }
   }
var car1 = Honda(color: .white, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)
var sportCar1 = HondaSport(color: .red, mp3: true, transmission: .manual, km:
0.0, doorState: .close, hatchState: .close)
```

Итак, у нас есть класс машины и мы можем выпускать необходимые автомобили. Но руководство нашей фирмы вновь требует нововведений. Они хотят выпускать специальную версию спорткара для выставочных стендов. Доработки минимальны: при попытке открыть люк посетитель должен получить уведомление, что этого нельзя делать.

Давайте создадим еще один подкласс, на этот раз будем наследоваться от спорткара. Так как метод открытия люка у нас уже есть, нам надо просто добавить в него функционал уведомления. Для этого воспользуемся возможностью переопределять методы родителя. Необходимо указать такое же имя метода и добавить перед ним ключевое слово «override». В самом методе вы можете вызвать реализацию родителя. Для этого необходимо обратиться к родителю через ключевое слово «super».

```
class DecorativeHondaSport: HondaSport {
                                        // переопределить метод открытия люка
    override func openHatch() {
        super.openHatch()              // вызываем реализацию родителя
        print("Пииип, открывать люк запрещено")// произносим уведомление
    }
}
```




Отлично, новые автомобили выпущены и установлены в залах для демонстрации. Но посетители открывают люки, несмотря на предупреждение. Руководство приняло решение заблокировать возможность открывать люк, но предупреждение оставить. Просто уберем из переопределенного метода вызов реализации родителя, и люк не будет открываться.

```
class DecorativeHondaSport: HondaSport {
    override func openHatch() {
        print("Пииииип, открывать люк запрещено")
    }
}
```

Вроде бы все пожелания учтены, но руководители фирмы опять нашли для нас работу. Они решили предоставить другой фирме права на выпуск нашего декоративного спорткара. Но они не хотят, чтобы эта фирма выпускала на его основе какие-то другие модели. К счастью, мы можем легко запретить создавать наследников от нашего класса. Просто добавим ключевое слово «final» перед объявлением класса. Наследоваться от final-классов запрещено. Кроме того, вы можете пометить «final» не весь класс, а только некоторые методы или свойства.

```
final class DecorativeHondaSport: HondaSport {
    override func openHatch() {
        print("Пииииип, открывать люк запрещено")
    }
}
```

Наследование – очень мощная возможность, но пользоваться ею следует с осторожностью. Наследование позволяет расширить возможности любого класса или изменить его поведение, но полностью удалить что-то из класса родителя вы не сможете.

# Деинициализация

В отличие от структур, классы имеют не только конструктор, но и деструктор. Правда, уничтожить класс деструктор не может – этим занимается среда выполнения. Зачем же он тогда нужен?

В Swift деструктор называется «deinit». Вы можете определить его у любого класса, но не можете его вызвать. Вызывает этот метод среда выполнения. Вы же можете описать в нем действия, которые необходимо выполнить в момент, когда объект удаляется из памяти.

Вернемся к моменту, когда мы добавляли в наш класс свойство для отслеживания количества выпущенных машин. Давайте превратим это свойство в счетчик существующих машин. Он будет отображать выпущенные, но не списанные автомобили.

```
class Honda {
    let color: UIColor
    let mp3: Bool
    let transmission: Transmission
    var km: Double
    var doorState: HondaDoorState
    // ключевое слово static указывает на то, что это свойство класса
    static var carCount = 0
    init(color: UIColor, mp3: Bool, transmission: Transmission, km: Double,
doorState: HondaDoorState) {
        self.color = color
        self.mp3 = mp3
        self.transmission = transmission
        self.km = km
        self.doorState = doorState
    // в конструкторе будем увеличивать переменную на 1
        Honda.carCount += 1
    }
    deinit { // в деструкторе уменьшим количество машин.
        Honda.carCount -= 1
    }
    static func countInfo(){
        print("Всего существует \(self.carCount) автомобиля")
    }
}
let car1 = Honda(color: .white, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)
var car2: Honda? = Honda(color: .white, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)
print(Honda.carCount) // 2
car2 = nil
print(Honda.carCount) // 1
```

Важно понимать, что вы не можете контролировать время удаления объекта из памяти – это решает среда выполнения. В большинстве случаев объект удаляется очень быстро, но надеяться на это нельзя.

# ARC и управление памятью

## Счетчик ссылок

Давайте еще раз посмотрим на пример с отслеживанием количества машин. В нем мы создали два объекта, затем переменной второго объекта присвоили nil, и у нас осталась только одна машина. Вы можете подумать, что мы явно удалили объект, присвоив ему «nil», но это не так.

Пройдемся немного по теории. В Swift действует механизм управления памятью под названием ARC – автоматический счетчик ссылок. Работает он просто: для каждого объекта класса ведется подсчет ссылок на него. Пока ссылок больше нуля, объект может спокойно существовать в памяти. Как только количество ссылок становится равным нулю, объект потенциально готов к удалению.

Давайте теперь разбираться, что такое ссылка. Это просто переменная, которая указывает на объект. И если говорить простым языком, пока на объект указывает хотя бы одна переменная, он будет существовать в памяти.

Простой пример. Создадим пять машин, для каждой из них будет своя переменная. Затем присвоим значение первой переменной остальным. В итоге у нас на первый объект будет ссылаться пять переменных – он будет иметь пять ссылок. На оставшиеся четыре объекта не будет ссылаться ни одна переменная, и они будут потенциально удалены.

```
// создадим 5 объектов, на каждый из них будет указывать одна ссылка
let car1 = Honda(color: .white, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)
var car2 = Honda(color: .white, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)
var car3 = Honda(color: .white, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)
var car4 = Honda(color: .white, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)
var car5 = Honda(color: .white, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)
Honda.carCount // существует 5 машин
// присвоим каждой переменной значение car1.
car2 = car1
car3 = car1
car4 = car1
car5 = car1
// теперь на первый автомобиль указывает 5 переменных
// на оставшиеся 4 автомобиля не указывает ни одна переменная.
Honda.carCount // существует 1 машина
```

Из примера видно, что счетчик машин равен единице. Это значит, что четыре объекта уже удалены. Это произошло сразу, как только мы убрали ссылку на них. Почему же я говорю, что объект **потенциально** готов к удалению? Дело в том, что как только счетчик ссылок опускается до 0, это означает, что разработчик больше не может взаимодействовать с объектом, но удалится он, только когда среда выполнения решит это сделать. Принимая решение, среда выполнения пытается оптимизировать работу с памятью и удаляет объект, когда посчитает нужным. В большинстве случаев это происходит почти мгновенно, но иногда может пройти значительно больше времени.

Вы можете спросить: какая нам разница, когда удаляется объект, если мы не можем с ним взаимодействовать? Можно представить, что он уже удален и выбросить его из памяти. В большинстве случаев так и следует поступать, но есть одно исключение: если вы воспользуетесь методом «deinit», чтобы отследить момент удаления объекта, и предположите, что он удалится, как только исчезнут все ссылки на него. Например, при создании объекта вы можете открыть файл или открыть канал связи с видеокамерой вашего телефона и закрыть файл или канал связи в методе «deinit». Тогда вы можете неприятно удивиться, обнаружив, что вроде бы уже удаленный объект удерживает ваши ресурсы заблокированными. Чтобы такого не случалось, создавайте специальный метод, в котором вы освободите ресурсы «closeFile» или «disconnectCamera», и вызывайте его явно, когда ресурсы вам больше не нужны.

Давайте посмотрим на еще один пример времени жизни объекта. Создадим пять машин, но четыре из них создадим внутри оператора «if». Как вы помните, внутри фигурных скобок создается новая область видимости. Все переменные, созданные внутри нее, автоматически уничтожаются, как только процесс выполнения программы покидает такую область. Так как переменные уничтожились, счетчик ссылок объектов, на которые они ссылались, стал равным нулю, и объекты уничтожились.

```
// создадим 1 объект
let car1 = Honda(color: .white, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)
let x = 10
// объявим if. Как вы помните, он создает новую область видимости
if x == 10 {
// внутри if создадим еще 4 объекта
    var car2 = Honda(color: .white, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)
    var car3 = Honda(color: .white, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)
    var car4 = Honda(color: .white, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)
    var car5 = Honda(color: .white, mp3: true, transmission: .auto, km: 0.0,
doorState: .close)
    Honda.carCount // существует 5 машин
}
// за пределами if
Honda.carCount    // существует 1 машина
```

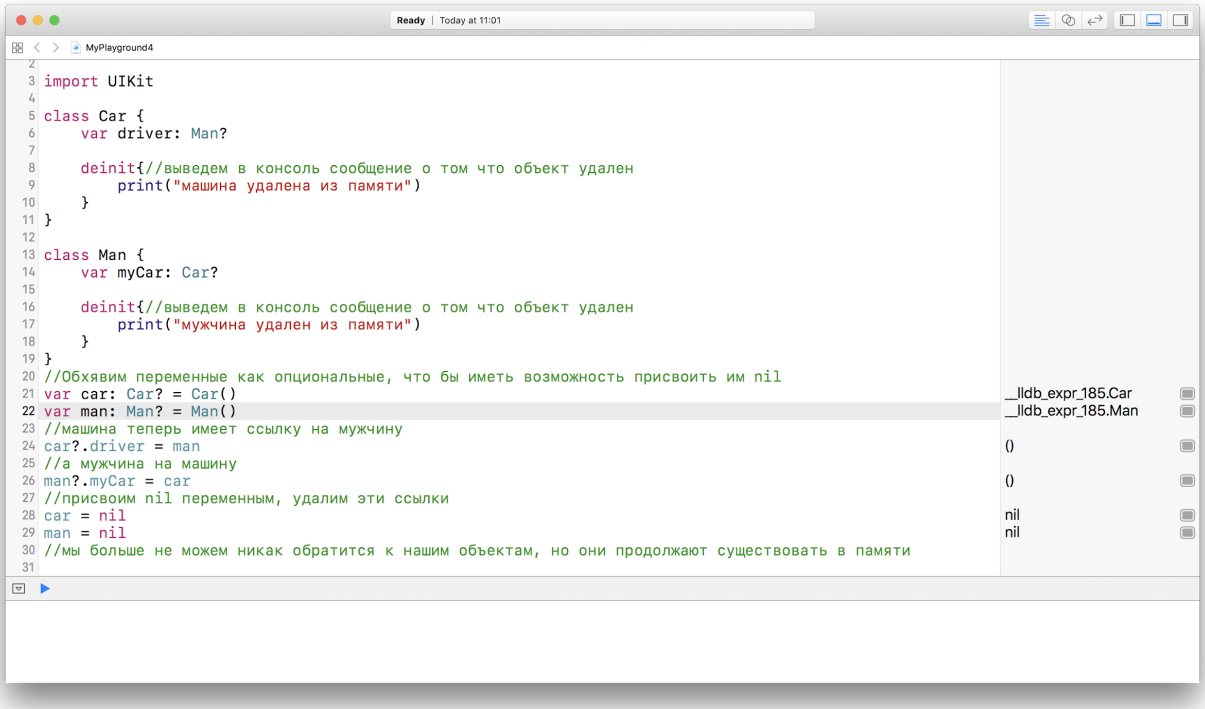
## Циклы удержания и слабые ссылки

Пока что процесс управления памятью выглядит очень простым и кажется, что никаких особых усилий прилагать не нужно. На самом деле часто в коде могут возникнуть ситуации утечек памяти.

**Утечкой памяти** называют ситуации, когда объект больше не доступен разработчику, но из памяти не удаляется. Такие ситуации возникают в результате циклов удержания. Самый простой цикл – когда объект А имеет ссылку на объект Б, а объект Б – ссылку на объект А.

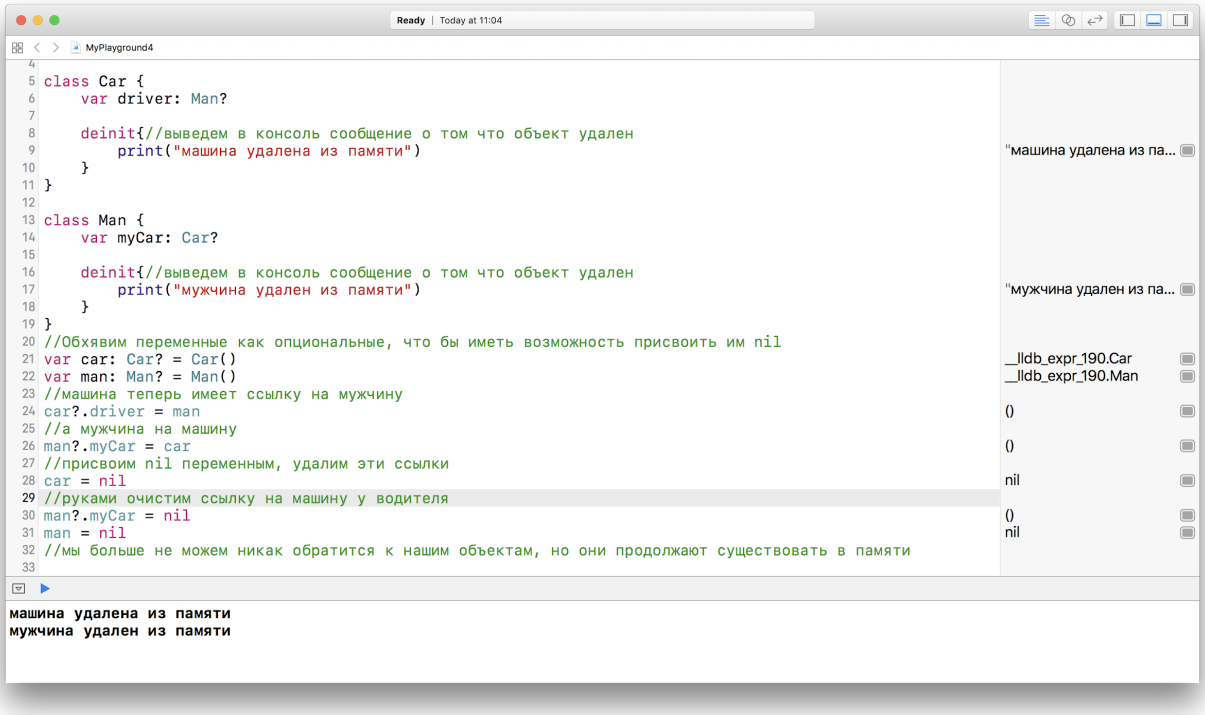
Давайте создадим новый класс машины с одним свойством, описывающим ее водителя, и класс мужчины со свойством, хранящим его машину. Свойства объявим как опциональные, чтобы установить их значение уже после создания объектов. Добавим обоим классам метод «deinit» и сделаем в нем вывод в консоль, чтобы отследить момент удаления из памяти. Инстанцируем объекты обоих классов, присвоим им ссылки друг на друга, после чего удалим другие ссылки на них.

```swift
class Car {
    var driver: Man?
    deinit{ // выведем в консоль сообщение о том, что объект удален
        print("машина удалена из памяти")
    }
}
class Man {
    var myCar: Car?

    deinit{ // выведем в консоль сообщение о том, что объект удален
        print("мужчина удален из памяти")
    }
}
// Объявим переменные как опциональные, чтобы иметь возможность присвоить им nil
var car: Car? = Car()
var man: Man? = Man()
// машина теперь имеет ссылку на мужчину
car?.driver = man
// а мужчина на машину
man?.myCar = car
// присвоим nil переменным, удалим эти ссылки
car = nil
man = nil
// мы больше не можем никак обратиться к нашим объектам, но они продолжают существовать в памяти
```



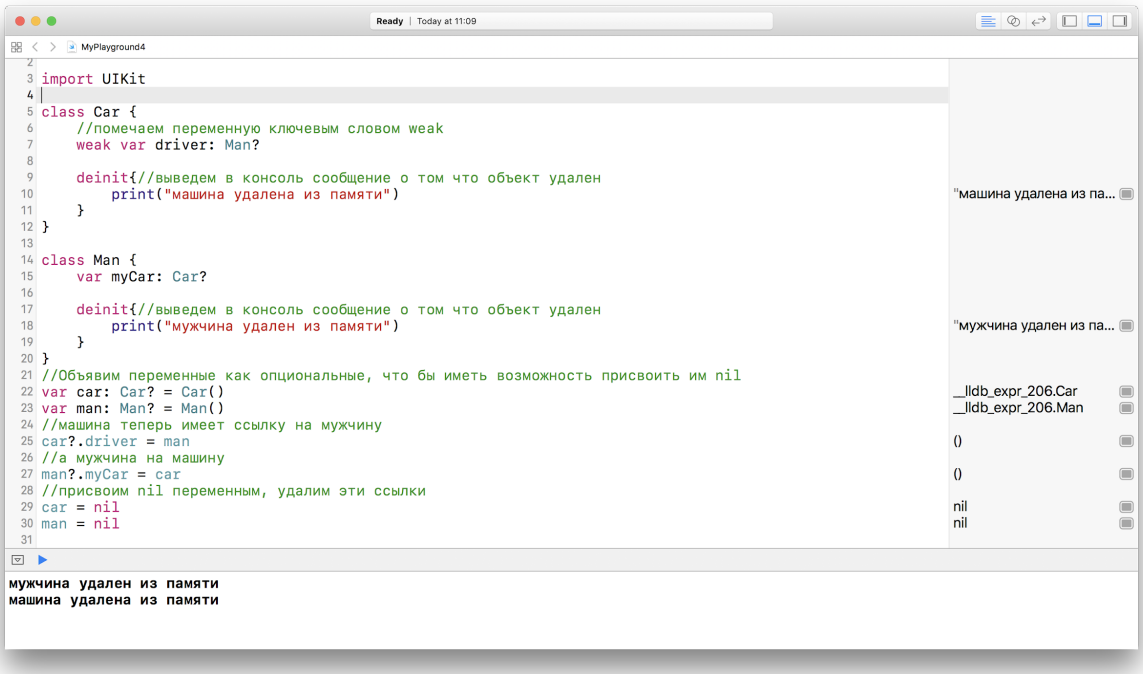

Как мы видим, никаких сообщений в консоль не вывелось, потому что объекты не удалились. Они ссылаются друг на друга, и их счетчик ссылок никогда не будет равен нулю.

Как же избежать такой ситуации? Конечно, вы можете сами удалить ссылку в одном из объектов.




Все сработало, сообщения в консоль вывелись, объекты удаляются из памяти, мы разорвали цикл удержания. Но это не очень удобно, ведь вам придется следить за каждым таким случаем удаления одного из объектов. Гораздо удобнее использовать так называемые «слабые ссылки». Слабые ссылки помечаются ключевым словом «weak» и не участвуют в процессе подсчета ссылок.

Давайте пометим переменную «driver» у класса «Car» как слабую и увидим, что объекты были удалены из памяти. Обратите внимание, что мы не сделали слабыми обе ссылки. Это сделано, чтобы, даже если извне не осталось переменных на машину, она не удалялась из памяти, пока жив ее владелец или пока он не поменяет машину.



Слабыми могут быть только изменяемые переменные («var»). Константы («let») обязаны содержать значение и не могут быть слабыми. Для них существует особый модификатор «unowned» – фактически то же самое, что и «weak», но для констант.

Например, у нас есть класс мужчины и его паспорта. Мужчина может родиться и не иметь паспорта, но паспорт выдается конкретному мужчине и не может выдаваться без указания владельца. Чтобы разрешить эту проблему, ссылку на паспорт у мужчины сделаем обычной опциональной, а ссылку на владельца у паспорта – «unowned» константой. Также добавим паспорту конструктор, чтобы сразу определить его владельца. Таким образом, человек сможет существовать без паспорта, сможет его поменять или выкинуть, но паспорт может быть создан только с конкретным владельцем и никогда не может его сменить. Тем не менее, цикла удержания мы избежали.

```
class Man {
    var myPassport: Passport?
    deinit{     // выведем в консоль сообщение о том, что объект удален
        print ("мужчина удален из памяти")
    }
}
class Passport {
    unowned let man: Man
    init(man: Man) {
        self.man = man
    }
    deinit{     // выведем в консоль сообщение о том, что объект удален
        print("паспорт удален из памяти")
    }
}
var man: Man? = Man()
var passport: Passport? = Passport(man: man!)
passport = nil // объект еще не удален, его удерживает мужчина
man = nil     // теперь удалены оба объекта
```

Важно понимать, что цикл удержания из двух объектов самый простой, его довольно легко заметить и избежать, но в цикле может участвовать 3, 6 и даже 10 и более объектов. Получится круг удержания, который вы можете не заметить никогда. Таких ситуаций поможет избежать только тщательное планирование архитектуры и иерархии связей между классами.

# Домашнее задание

1. Описать класс Car с общими свойствами автомобилей и пустым методом действия по аналогии с прошлым заданием.
2. Описать пару его наследников TrunkCar и SportCar. Подумать, какими отличительными свойствами обладают эти автомобили. Описать в каждом наследнике специфичные для него свойства.
3. Взять из прошлого урока enum с действиями над автомобилем. Подумать, какие особенные действия имеет TrunkCar, а какие – SportCar. Добавить эти действия в перечисление.
4. В каждом подклассе переопределить метод действия с автомобилем в соответствии с его классом.
5. Создать несколько объектов каждого класса. Применить к ним различные действия.
6. Вывести значения свойств экземпляров в консоль.

# Дополнительные материалы

1. Официальная документация.
2. Брюс Эккель. Философия Java.

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://developer.apple.com/library/prerelease/content/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html#//apple_ref/doc/uid/TP40014097-CH5-ID309.