Разработка под iOS. Начинаем

Часть 1

Swift как язык программирования

Оглавление

Глава 1

Разработка под iOS. Начинаем

1.1. Обзор экосистемы Swift

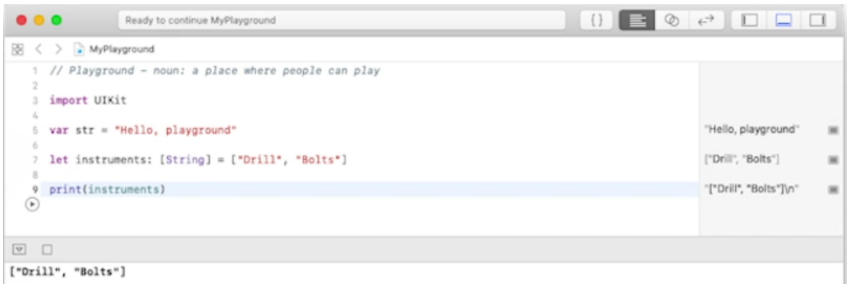
1.1.1. О языке Swift

Итак, платформа iOS появилась 9 января 2007 года. Была представлена лично Стивом Джобсом на выставке-конференции Macworld Conference&Expo и выпущена в июне того же года.
Первым языком, на котором можно была писать программы под iOS, был, что характерно, JavaScript. Под самый первый iOS не было нативных SDK, и единственным, что можно было сделать под первый iPhone, были web-приложения для браузера Safari.
Время шло, со второй версией iOS Apple выпустила нативный SDK - Cocoa Touch. Языком разработки нативных приложений стал Objective-C, на котором к тому времени уже писались программы для macOS (которая тогда называлась OS X).
И, по историческим меркам, совсем недавно появился язык Swift. С момента его официальной выкладки в Open Source он стал основным языком разработки прикладных приложений под Apple'овские платформы.
Swift уже пережил ряд мажорных версий. Первая версия языка Swift кардинально отличалась от того, что мы можем видеть сейчас. После этого появилась вторая версия, на которой уже можно было нормально разрабатывать прикладные приложения, и которая была выложена в Open Source. Затем была представлена третья версия, которая привнесла много нового, в основном большое переименование всего и вся. Этот процесс переименования даже получил свое собственное название – "Великое Переименование" или Grand Renaming. И затем еще через год вышла четвертая версия, которая по большей части является улучшением и полировкой самого языка. *Ждем пятую версию, в которой, возможно, наконец-то, появится бинарная совместимость (ABI Stability)*.
Несмотря на то, что Swift изначально планировался как язык разработки для платформ компании Apple, в момент выкладки его в Open Source Apple решила не останавливаться только на

своих операционных системах, но и сделать версию компилятора и стандартной библиотеки для операционной системы Linux. Изначально это был только компилятор и стандартная библиотека, но затем на этой базе сторонние фирмы, в том числе, например, IBM начали разработку различных web-фреймворков. На данный момент можно сказать, что на Swift можно писать web-приложения, т.е. бэкенды для них, и даже выкладывать их в продакшн.
Таким образом, с учетом server-side'а можно считать, что Swift есть практически везде. На самом деле, компилятор и стандартная библиотека были портированы под Windows и под Android. Вы можете собирать бинарные файлы как под Windows, так и под Android. Вопрос только в том, что UI-компонентов и библиотек под эти системы нет. Тем не менее, Swift как язык есть везде.
Так как на момент выхода Swift'а основным языком разработки под Apple'овские платформы был Objective-C, Apple не стала "рубить с плеча" и затевать революций, как это в свое время было сделано в OS X, когда нативный фреймворк Carbon был без обратной совместимости заменен нативным фреймворком Cocoa, который в отличии от Carbon был на Objective-C, а был разработан специальный слой interoperability между Objective-C и Swift'ом. Таким образом, в вашем приложении вы можете относительно легко комбинировать код, написанный на Objective-C и на Swift, и даже на чистом C. Вы можете осуществлять плавный перевод вашей кодовой базы с Objective-C на Swift без необходимости делать это разово или единомоментно.
Надо, однако, обратить внимание, что с выходом версии языка Swift 3.0, произошел процесс "великого переименования" и привычные длинные наименования методов из Objective-C стали гораздо короче и лаконичнее. Это стоит помнить, тогда когда мы занимаемся разработкой в рамках одного приложения и на Swift, и на Objective-C.

1.1.2. Инструменты для разработки на Swift

Для того, чтобы что-нибудь на языке Swift написать, нам нужны инструменты. При этом используемые инструменты зависят от того, под какую платформу вы хотите писать прогрыммы: будь это эпловские платформы, или вы хотите написать какой-нибудь server-side под Linux, или вы хотите просто поиграться.
Начнем с "поиграться". Вам понравился язык Swift, вы хотите с ним поэкспериментировать. Для того, чтобы начать, вам достаточно какого-нибудь online-инструмента, коих сейчас есть великое множество, которые позволяют вам не выходя из браузера экспериментировать с языком и его стандартной библиотекой.
Если этого вам недостаточно, но вы по-прежнему хотите пока только изучать язык без написания серьезных приложений, тогда для вас в среде разработки Xcode есть инструмент, который называется Swift Playgrounds. Он позволяет вам фактически в режиме блокнота с автоподсветкой синтаксиса, некоторой документацией и автоматической компиляцией экспериментировать как с самим языком и стандартной библиотекой, так и с библиотеками для разработки под iOS или macOS.



Если же вы хотите уже не просто поиграться, а заняться разработкой приложения, пусть даже простого, под iOS или macOS, под часы или TB-приставку, тогда вам уже потребуется базовый набор инструментов для разработки под эпловские платформы. Сюда входит компьютер под управлением macOS. Под эпловские платформы нельзя разрабатывать ни на каких других компьютерах. Поэтому вам так или иначе потребуется эпловский компьютер с операционной системой macOS. Вам потребуется среда разработки Xcode. И если вы хотите выкладывать ваши приложения в AppStore, то вам потребуется оплаченный аккаунт разработчика. Если вы хотите пока просто разрабатывать, вам будет достаточно бесплатного аккаунта. Также вам потребуется устройство (iPhone, iWatch, ...) для отладки ваших приложений на реальных устройствах.
Важно отметить, что Xcode – это практически единственная среда разработки, которая дает вам полный спектр инструментов для разработки под эпловские платформы. Существует еще среда разработки AppCode от JetBrains, которая также позволяет разрабатывать под, например, iOS, но при этом, например, Interface Builder вам будет недоступен. Имейте это ввиду.
Ну и наконец, для server-side разработки вы также можете использовать эпловские платфомы и среду разработки Xcode. Однако в момент выкладки в продакшн вам так или иначе потребуется компьютер с операционной системой Linux, на котором установлены пакеты swift-компилятора и стандартной библиотеки. Однако для конкретно server-side разработки вам будет достаточно какой-нибудь среды разработки прямо на Linux'е. Например, тот же CLion от JetBrains, в котором есть необходимые инструменты.
Кроме инструментов для разработки вам потребуется документация. Основным источником документации является developer.apple.com, на котором расположена документация как по самому языку и его стандартной библиотеке, так и по библиотекам для разработки под iOS, macOS и т.д.

developer.apple.com

Кроме этого, некоторую документацию вы можете найти на сайте swift.org. Это основное место жительства языка Swift в Open Source, где есть необходимые ссылки, необходимая информация об процессе разработки, процессе эволюции языка. Там же вы можете найти proposals – предложения по улучшению языка, которые предлагают разные разработчики и компании. Там же вы можете найти ссылки на дистрибутивы пакетов для macOS и Linux.

swift.org

Кроме этого, есть книжка про Swift, называется Swift Book. Это такая своего рода энциклопедия языка. Есть ее русская версия.

swiftbook.ru

Так как язык Swift выложен в Open Source, есть проект на github – github.com/apple/swift, где вы можете ознакомиться с исходным кодом самого компилятора и стандартной библиотеки. Иногда очень полезная штука, когда есть какая-то проблема, и вы хотите досконально в ней разобраться.

github.com/apple/swift

Есть также рассылка, на которую можно подписаться, чтобы быть в курсе последних веяний в языке, процессов разработки, вносимых proposals и т.д.
Ну и есть так называемый swift evolution process - это такой процесс, который направлен на улучшение экосистемы языка. В нем участвуют разные люди и разные компании, которые могут вносить свои предложения. Предложения могут быть рассмотрены и реализованы в дальнейшем.
На этом вступительная часть закончена. Давайте перейдем к непосредственному рассмотрению языка.

1.2. Базовые конструкции языка

1.2.1. Базовые конструкции языка

Давайте перейдем к базовым конструкциям языка swift. Первым и, пожалуй, самым невинным отличием от родственных языков является отсутствие необходимости в конце каждого выражения ставить точку с запятой. Достаточно писать законченное выражение вот таким вот образом:

```
print("Hello, World")
```

Однако если вы хотите разместить два выражения на одной строчке, то точка с запятой в этом случае вам понадобиться:

```
print("Hello") ; print("World")
```

Теперь перейдем к объявлению констант и переменных. Для того, чтобы создать константу, вам потребуется ключевое слово let, имя вашей переменной и ее значение. В языке swift тип может быть выведен компилятором, и обычно опускается. Если тип не задан, компилятор попытается вывести его из заданного первоначального значения:

```
let a = 10
```

В данном примере компилятор для переменной a выведет тип Int. Если вы хотите создать переменную, тогда вместо ключевого слова let вам потребуется ключевое слово var, а остальное не

поменяется:

```
var b1 = UIView()
b1 = UIView()
```

Также иногда существует необходимость задать тип вашей переменной или константе. Для этого его надо указать вот таким вот способом:

```
let a: Int = 10
var b: UIView = UIView()
```

Теперь давайте перейдем к ветвлениям. В языке swift, как и в большинстве языков программирования, существует конструкция if-else:

```
var something = true
if something {
    // делаем что-нибудь
} else {
    // делаем что-то другое
}
```

И в отличие от родственных языков, таких как C/C++, в swift отсутствует необходимость обрамлять условие в круглые скобки, если они там явным образом не нужны (например, для задания приоритета операций). Swift поддерживает стандартные конструкции if-elseif-else. Примерно таким образом:

```
var something2 = false
if something {
    // делаем что-нибудь
} else if something2 {
    // делаем что-нибудь не то
} else {
    // делаем что-то другое
}
```

Что же касается циклов, то в отличие от C/C++, в языке swift нет стандартного цикла for. Он когда-то существовал в swift, но впоследствии был упразднен, и был заменен более продвинутым вариантом на базе ranges вот так вот:

```
for i in 0..<10 {
    print("i: ", i)
}
```

```
for i in 0...10 {
    print("i: ", i)
}
```

В первом варианте индекс пройдет от 0 до 9, во втором – от 0 до 10.
Цикл while выглядит привычным образом. Следует еще раз обратить внимание на то, что скобки явным образом не нужны. Во всем остальном это привычный while:

```
var i1 = 0
while i1 < 10 {
    print("i1: ", i1)
    i1 += 1 // в Swift нет конструкции i1++
}
```

Также стоит обратить ваше внимание на то, что привычной постфиксной и префиксной записи инкремента в единицу в swift нет, такой вариант инкремента и декремента был упразднен:

```
var a3 = 1
// нет a3++
a3 += 1
```

Цикл while с постусловием выглядит немного непривычно. Ключевое слово ”do”, привычное по другим языкам, заменяется на ключевое слово ”repeat”. Постусловие выглядит также, скобки не нужны, точка с запятой в конце также не нужны. Главное не забываем про инкремент:

```
var i2 = 0
repeat {
    print("i2: ", i2)
    i2 += 1
} while i2 < 10
```

Существует еще цикл foreach, который позволяет итерировать по коллекциям, выглядит он так:

```
let arr = [1, 2, 3]
for element in arr {

}
```

К нему мы вернемся чуть позже, когда будем рассматривать коллекции.
В Swift имеет место обработка исключений. Как и в других языках, функции могут генерировать исключения, и их необходимо обрабатывать. Для этого существует конструкция do-try-catch:

```
do {
    try doSomething()
} catch {
    print("e: \(error)")
}
```

Как объявить функции, которые могут генерировать исключения, и как сгенерировать исключение, рассмотрим в разделе функций.

1.2.2. Функции

Теперь давайте перейдем к функциям. Вообще, в любом языке программирования функции - базовый примитив для структурирования вашего кода. Вы можете оформлять какие-нибудь блоки кода в функции, и затем их переиспользовать в разных местах вашей программы.
Функции в языке Swift тоже есть. Давайте рассмотрим как они объявляются и как они используются. Начнем с самого базового варианта - функция, которая не принимает на вход никаких параметров и ничего не возвращает, или иначе процедура. Записывается она вот так:

```
func foo1() {
    // не делаем ничего :(
}
```

Теперь давайте рассмотрим функцию, которая является уже собственно функцией, и которая возвращает некоторое значение, например, целочисленное. Тогда мы объявляем ее вот таким вот образом:

```
func foo() -> Int {
  return 1
}
```

Однако функций, которые просто возвращают значение и процедуры, при разработке программ недостаточно. Большая часть ваших функций так или иначе будет принимать на вход некоторые параметры.
Важно здесь отметить, что так как Swift в части определения параметров функций, определенные идеи и соглашения взял из Objective-C, то у каждого входного параметра функции есть так называемые внутреннее и внешнее имя. Внутреннее имя - это то, как этот параметр называется внутри тела функции, а внешнее имя - это то, как этот параметр называется для того, кто функцию вызывает.
Часто внутреннее и внешнее имя параметра совпадают. Таким образом, имя параметра для того, кто функцию вызывает, и внутри тела самой функции будут одинаковыми. Например, вот так:

```
func say1(p: String) {
    print(p)
}
```

И вызываться она будет вот так:

```
say1(p: "Hello, World")
```

Иногда бывают моменты, когда мы хотим, чтобы для внешнего пользователя функции параметр назывался одним именем, а внутри функции использовался под другим именем. Например, мы хотим сократить при использовании длинное внешнее имя, или мы работаем с Objective-C и у нас есть некоторый параметр с внешним именем withSomething. Не очень читабельно будет использовать внутри функции это имя, проще написать something, и таковым будет его внутреннее имя. Записывается это вот таким вот образом:

```
func say2(phrase p: String) {
    print(p)
}
```

В данном примере внутреннее имя будет "p", и внутри функции будет использовать параметр под этим именем. Но при вызове для внешнего наблюдателя этот параметр уже будет называться "phrase". Соответственно, вызов будет выглядеть вот таким вот образом:

```
say2(phrase: "Hello, World")
```

Внешнее имя может отсутствовать, тогда записывается это вот так:

```
func say3(_ p: String) {
    print(p)
}
```

И в этом случае вызов такой функции для вызывающей стороны будет выглядеть как традиционный во многих языках вызов функции вообще без имени параметра, вот таким вот образом:

```
say3("Hello, World")
```

Во многих языках, и Swift не исключение, при объявлении функции можно задать для ее параметров некоторые значения по умолчанию. Например, мы объявили функцию sum, которая принимает два целых числа и возвращает их сумму, и решили не задавать никаких значений по умолчанию. Это будет выглядеть так:

```
func sum(lhs: Int, rhs: Int) -> Int {
    return lhs + rhs
```

```
}
```

И вызываться будет только так:

```
print(sum(lhs: 1, rhs: 1))
```

А для функции вычитания sub мы решили задать параметры значения по умолчанию:

```
func sub(lhs: Int = 0, rhs: Int = 0) -> Int {
    return lhs - rhs
}
```

И теперь мы можем вызвать функцию sub вот так:

```
print(sub())
```

Тогда она резонно вернет 0. Или вот так:

```
print(sub(lhs: 1))
```

Тогда мы из 1 вычтем 0, который задан по умолчанию для второго параметра, и получим 1. Или вот так:

```
print(sub(rhs: 1))
```

Тогда мы из 0 по умолчанию для первого параметра вычтем 1 и получим -1. И, наконец, вот так:

```
print(sub(lhs: 1, rhs: 1))
```

Тогда мы из 1 вычтем 1 и закономерно получим 0.
Кроме того, Swift для удобства программирования предлагает ряд интересных конструкций. Например, конструкция guard. Guard позволяет вам написать некоторый оператор if, но с инверсной логикой. Звучит это примерно следующим образом: ”Я хочу, чтобы это условие выполнялось, иначе сделай какой-то блок кода”. Например, вернуть ошибку.
Это очень помогает, когда вам на входе в функцию надо проверить значения входных параметров, что пользователь передал их в нужном диапазоне значений и в нужном формате, и если это не так, что-то сделать. Это своего рода assert, но который не приводит к крэшу приложения, а позволяет вам выполнить некоторый блок кода, если условие не выполняется.
Записывается это так. Например, у нас есть некоторая функция:

```
func two(a: Int) -> String? {
```

И мы хотим проверить, что введенное значение есть 2, а если нет - то вернуть nil:

```
func two(a: Int) -> String? { // тут String? со знаком вопроса - это правильно;
это Optional, которые будут рассмотрены позднее
    guard a == 2 else {
        return nil
    }
    return "two"
}
```

Соответственно, с помощью оператора guard очень удобно реализовывать так называемый золотой путь (golden path), или счастливый путь (happy path), когда вы не делаете огромное количество вложенных проверок, а проверяете их сразу, и возвращаете ошибку, если они некорректны.

В дополнении к guard для удобства работы с функциями есть конструкцию defer, которая помогает вам не "прострелить себе ногу" в случаях, когда у вас очень много возвратов из функции, при которых необходимо сделать некоторое действие.

Например, у нас есть функция, которая открывает дверь, и для этого делает ряд каких-то действий. И если какое-то из этих действий не выполняется, то мы немедленно закрываем дверь и выходим.

```
func openTheDoor_bad(condition1: Bool, condition2: Bool) {
    let door = Door()
    door.open()

    if condition1 {
        door.close()
        return
    }

    // тут много разного кода
    if condition2 {
        // тут забыли сделать door.close()
        return
    }

    // тут много разного кода
    door.close()
}
```

В каком-то из условий мы забыли вызвать функцию закрытия двери (door.close). Таким образом,

мы можем получить ситуацию, когда дверь останется открытой. Поэтому, дабы сделать код корректным и уменьшить вероятность ошибки, мы можем добавить конструкцию defer, а все вызовы door.close в условиях возврата убрать:

```
func openTheDoor_good(condition1: Bool, condition2: Bool) {
    let door = Door()
    door.open()
    defer { door.close() }

    if condition1 {
        // тут не надо door.close()
        return
    }

    // тут много разного кода
    if condition2 {
        // тут не надо door.close()
        return
    }

    // тут много разного кода
    // тут не надо door.close()
}
```

Таким образом, если мы где-то забыли вызвать функцию door.close, или впоследствии при развитии кода написали еще одно условие возврата, где опять-же могли забыть вызвать функцию закрытия дверей, у нас не возникнет такой проблемы, так как конструкция defer при возврате из функции сделает это дело за нас.
Как уже говорилось выше, в Swift реализована поддержка генерирования и обработки исключений. Как обработать исключения мы уже рассмотрели ранее. Теперь давайте рассмотрим как реализовать функцию, которая может генерировать исключения.
Для этого достаточно добавить к ее сигнатуре конструкцию throws:

```
func doSomething() throws -> Void {
}
```

Для того, чтобы сгенерировать исключение, нам необходим тип. Лучше всего для этой цели использовать перечисление с базовым типом Error:

```
// Перечисления (enum) будут детально рассмотрены позднее
enum OurErrors: Error {
    case somethingBadHappened
}
```

И теперь мы можем в нашей функции сгенерировать исключение:

```
func doSomething() throws -> Void {
    throw OurErrors.somethingBadHappened
}
```

При вызове данной функции это исключение надо будет обработать. На этом с функциями все. Перейдем к Optionals.

1.2.3. Optionals

Теперь давайте рассмотрим еще один интересный момент языка Swift, так называемые Optionals. Константы и переменные могут быть могут быть Optional, то есть иметь значение не всегда, и non-Optional, то есть иметь значение всегда.
Обычные константы и переменные, как например, вот эта:

```
var thing: String = "Hello"
```

содержат значения всегда. Однако в практике программирования существуют случаи, когда требуется переменная, которая в некоторых случаях может не содержать значения. И тогда здесь нам помогут optionals. Выглядит это вот таким вот образом:

```
var maybeThing: String? = nil
maybeThing = "Hello"
```

И говорит о том, что переменная maybeThing может либо содержать некоторое значение типа String, либо его не содержать, и тогда значение будет nil.
Соответственно, когда мы работаем с обычными константами и переменными, то их значение, или значение их полей, если это объекты, как, например, UIView, мы получаем через точку, как это применимо во многих языках:

```
var view1 = UIView()
print(view1.frame)
```

Однако когда мы говорим об Optional, мы не можем напрямую получить значение потому, что, строго говоря, Optional есть некоторая структура-обертка над типом, и вот такая запись:

```
var view2: UIView?
```

есть синтаксический сахар для этой специальной структуры. Эта запись эквивалентна вот такой:

```
var view3: Optional<UIView>
```

Поэтому для того, чтобы извлечь значение переменной или значения ее полей, например, поля frame, мы можем воспользоваться одним из двух вариантов. Первый вариант с проверкой - с помощью записи "?.":

```
var frame2 = view2?.frame
```

Когды мы используем такую запись, мы достаем значение с проверкой на то, присутствует это значение, или нет. Если переменная не содержит значения, то есть содержит nil, то результатом этой операции также будет значение nil. Если же значение присутствует, то мы его получим обернутым в Optional.
Можно пойти другим путем. Вариант без проверки - с помощью записи "!.":

```
var frame3 = view!.frame
```

Если значение в этой переменной есть, то мы получим его как non-optional, без оборачивания его в Optional. Иногда это полезно, когда вы хотите получить значение и не хотите больше работать с optional. Такой подход называется Force Unwraping Optionals, или, своего рода, силовое разыменование optionals.
При таком подходе мы говорим компилятору "мы знаем что мы делаем, там действительно есть значение, просто дай мне его".
Почему такой подход в целом плох несмотря на то, что есть случаи, когда он просто необходим, давайте рассмотрим на примере. У нас есть два объекта класса UIView. Один из них не будет содержать значения и nil, второй будет содержать некое значение:

```
var view4: UIView? = nil
var view5: UIView? = UIView()
```

Теперь, когда мы попытаемся получить значение поля frame через операцию разыменования с проверкой, то в первом случае мы получим nil, так как мы проверили наличие значения, и его там не оказалось. Во втором случае, так как там есть значение, мы его получим обернутым в Optional:

```
var f1 = view1?.frame // nil
var f2 = view2?.frame // Optional<frame>
```

Однако если мы попытаемся force unwrap'ить это дело, тогда в первом случае мы получим runtime crash приложения, с которым мы уже ничего не сможем сделать. Приложение у поль-

зователя аварийно завершиться. Во втором случае мы получим значение frame'а в чистом виде (без optional):

```
var f1 = view1!.frame // CRASH
var f2 = view2!.frame // frame
```

Поэтому нам нужны подходы для того, чтобы получать значения, и по возможности не получать runtime crash'ы, потому что это так или иначе влияет на восприятие нашего приложения пользователями. И первое, что нам в этом поможет - это конструкция if let:

```
if let view = view {
}
```

Если view - это optional view, то внутрь ветвления мы попадем только если там есть значение, и внутри цикла переменная view уже не будет optional. Конструкция достаточно простая. Теперь, возвращаясь к нашему примеру с двумя optional:

```
var view1: UIView? = nil
var view2: UIView? = UIView
```

в первом случае внутрь цикла мы не попадем

```
if let v1 = view1 {
}
```

так как во view1 у нас лежит nil, а во втором случае:

```
if let v2 = view2 {
}
```

Мы получим разыменование значения, пригодное для дальнейшего использования в константу v2. Сразу хочу отметить, что это самый лучший подход в части работы с optional, потому что здесь вы и проверяете на наличие значения, а когда вы делаете переменные optional, там может не быть значения, и это надо обязательно проверять. Так и получаете значение, очищенное от optional.
Примерно в таком же виде можно использовать и конструкцию guard:

```
guard let v = view else { ... }
```

Что очень удобно для проверки и разыменования optionals на входе в функцию:

```
func getFrame(view: UIView?) throws -> CGRect {
guard let v = view else {
return.zero
}
```

```
return v.frame
}
```

Если значение будет отсутствовать, мы получим ошибку, которую можно обработать. Иначе получим валидный frame.
Здесь еще раз хочется сказать про Golden, или Happy Path, о котором я говорил чуть ранее. Чтобы вместо монструозных конструкций писать более элегантный код. Например, приведенную выше функцию можно было бы написать некрасиво так:

```
func getFrame(_ view: UIView?) throws -> CGRect {
if let v = view {
return v.frame
} else {
throws return.zero
}
}
```

Согласитесь, что первый вариант с конструкцией guard выглядит гораздо более понятно.

1.3. Объектно-ориентированное программирование

1.3.1. ООП

Как уже было сказано выше, Swift – объектно-ориентированный язык программирования. Поэтому как в полноценном объектно-ориентированном языке в нем присутствует понятие класса. Чтобы создать новый класс, достаточно его объявить:

```
class Car {
}
```

Однако пустой класс относительно бесполезен. Для того, чтобы его хоть как-то можно было использовать, ему нужны поля – константы и переменные. Константы класса мы можем объявлять так же, как и глобальные, через ключевое слово let:

```
class Car {
let mark: String
let model: String
}
```

Соответственно, поля-переменные можно объявлять также, как и глобальные, вот так:

```
class Car {
...
var mileage: Int
}
```

В определенной части объектно-ориентированных языков существуют так называемые статические (классовые) поля. Такие есть и в Swift'e. Для того, чтобы их объявить, необходимо воспользоваться либо ключевым словом class (let или var), либо ключевым словом static (let или var). Так уже повелось в Swift, что для этих целей эти два ключевых слова – взаимозаменяемы. Вы можете использовать то, которое вам больше нравится. Обычно для констант используют ключевое слово static, а для переменных – class. Выглядит это так:

```
class Car {
...
static let steelType: Int
class var productionAmount: Int
}
```

Для получения доступа к полям класса необходимо воспользоваться точечной нотацией, также как это принято в основных объектно-ориентированных языках:

```
let car: Car = Car()...
print(car.model)
print(car.mark)
car.mileage += 1
```

Таким же образом как мы объявляем поля, можно объединить и функции – воспользоваться теми же способами, как мы объявляем глобальные функции:

```
class Car {
...
func drive(to Location: CLlocation) {
}
```

Как и поля, функции могут быть функциями экземпляра (принадлежать конкретному экземпляру класса), либо функциями класса (принадлежать самому классу). Обычные функции экземпляра объявляются просто как функции. А функции класса определяются через ключевое слово static (или class):

```
class Car {
...
static func make() -> Car {
```

```
return Car()
}
```

Важно отметить вот какой аспект. Когда вы объявляете поле через ключевое слово var, как я показал выше, то вы создаете так называемое хранимое поле. Поле, которое в области памяти, выделенном под экземпляр класса, будет занимать некоторое (согласно типу) место. Однако Swift позволяет определять так называемые вычислимые поля, которые для внешнего наблюдателя выглядят как поля, но на самом деле являются одной или двумя функциями, и памяти в области экземпляра не занимают. Для того, чтобы объявить вычислимое свойство, необходимо воспользоваться тем же ключевым словом var, но при этом задать функции setter и getter:

```
class Car {
...
var buyDate: Date
var buyTimestamp: TimeInterval {
set { buyDate = Date(timeIntervalSince1970: newValue) }
get { return buyDate.timeIntervalSince1970 }
}
}
```

Однако если вы хотите вычислимое свойство только для чтения, вы можете опустить функцию setter, а если у вас есть только геттер, то можно опустить и объявление дополнительной функции. Таким образом, вычислимые свойства только для чтения могут выглядеть так:

```
class Car {
...
var buyDate: Date
var buyTimestamp: TimeInterval {
return buyDate.timeIntervalSince1970
}
}
```

Как уже выше говорилось, Swift – это про краткость, и в то же время про емкость выражений. Как и в других объекто-ориентированных языках, в Swift есть модификаторы доступа. Здесь вам доступны такие модификаторы как public; private; private(set) – который внутри класса работает как public, а во вне дает доступ только для чтения; fileprivate – который внутри файла с исходным кодом работает как public, то есть для всех объектов в рамках одного файла исходного кода работает как public, а для других файлов – как private. Для того, чтобы воспользоваться модификатором доступа, его необходимо добавить перед объявлением поля или функции:

```
class Car {
    private let model: String
public var mileage: Int
private(set) var guardian: String = "Sentinel"
fileprivate var ownedBy: UUID
}
```

Важно отметить, что если вы не объявили модификатор доступа, то он получит значение internal. Он работает так. Внутри конкретного модуля (бинарника приложения или библиотеки) он работает как public, в том числе на изменение. Для всех объектов, которые находятся вне вашего модуля (например, вы разрабатываете библиотеку), он будет работать как private.
Исторически так получилось, что в Swift нет таких понятий как конструкторы и деструкторы. Так как Swift идеологически вырос из Objective-C, то он наследует идеи инициализации и деинициализации из Objective-C. В котором при создании экземпляра класса вы делаете так:

```
Car* car = [[Car alloc] init];
```

Сначала вызывается функция alloc, которая выделяет память под ваш объект нужного размера. А затем в эту память вызывается инициализатор, который инициализировал эту память нужными значениями. Поэтому вместо конструкторов и деструкторов в Swift объявляются инициализаторы и деинициализаторы. Во всем остальном можно рассматривать инициализаторы как конструкторы (также необходимо вызывать конструктор суперкласса), а деинициализаторы как деструкторы (не требуется вызывать деструктор суперкласса). Инициализаторы объявляются так:

```
class Car {
...
init(mark: String, model: String) {
self.mark = mark
self.model = model
super.init()
}
}
```

А деинициализаторы так:

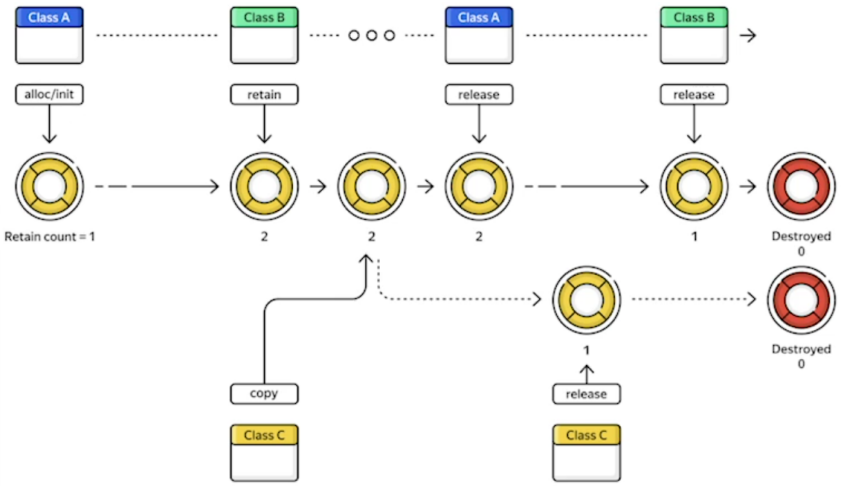
```
class Car {
...
deinit {
Press.crash(self)
}
```

```
}
```

В Swift существует понятие главных (designated) инициализаторов, то есть базовые наиболее общие инициализаторы, задача которых полностью инициализировать объект, и вспомогательных (convinience) инициализаторов, которые могут добавлять какие-то специальные сигнатуры, но при этом внутри таких инициализаторов необходимо вызвать базовый.

1.3.2. Управление памятью

А теперь давайте поговорим про управление памятью в программах на Swift. Для того, чтобы разобраться как устроено управление памятью на Swift, необходимо обратиться к истокам и рассмотреть как было устроено управление памятью на Objective-C.
Изначально в objective-c существовало ручное управление памятью на основе подсчета ссылок (или MRC - manual reference counting). Выглядело это так. У каждого объекта наследника NSObject имеется скрытый счетчик ссылок, который увеличивался тогда, когда кто-то заявлял о своем владении объектом, и уменьшался, когда кто-то из владельцев заявлял о прекращении владения объектом.
Таким образом, у объекта были следующие функции, которые влияли на счетчик ссылок. При создании объекта с помощью функции alloc, он создавался с счетчиком ссылок, равным единице. Если стороннему объекту требовалось заявить о владении, необходимо было вызвать функцию retain, которая увеличивала счетчик ссылок. Также можно было вызвать функцию copy, которая создавала копию объекта с счетчиком ссылок равным единице. Когда объект переставал быть нужным, требовалось вызывать функцию release, которая уменьшала счетчик ссылок, и когда счетчик ссылок доходил до нуля, объект удалялся. Еще была такая замечательная функция autorelease, которая работала примерно также, как release, но отложенно.
Давайте рассмотрим, как это все работало на примере вот такой картинки:



Некий класс A, который создает наш объект с помощью вызова функции alloc. Счетчик ссылок будет равным единице. Далее некий класс B, также хочет владеть экземпляром объекта, поэтому он вызывает функцию retain. Теперь на наш объект ссылаются два класса и счетчик ссылок у него равен двум. Класс C рассмотрим чуть позже. Далее класс A закончил работу с объектом, и он ему более не нужен, он вызывает функцию release, и счетчик ссылок уменьшается на единицу и становится равным единице. Далее для класса B также решаем, что объект ему не нужен, и тоже вызывает функцию release (или например autorelease), счетчик ссылок уменьшается до нуля, объект удаляется. Если где-то в ходе этого процесса некому классу C понадобилась копия нашего объекта, он вызывает функцию copy, получает копию объекта с счетчиком ссылок равным единице. И когда он вызовет функцию release, то счетчик ссылок копии объекта станет равным нулю, и копию объекта будет уничтожена.

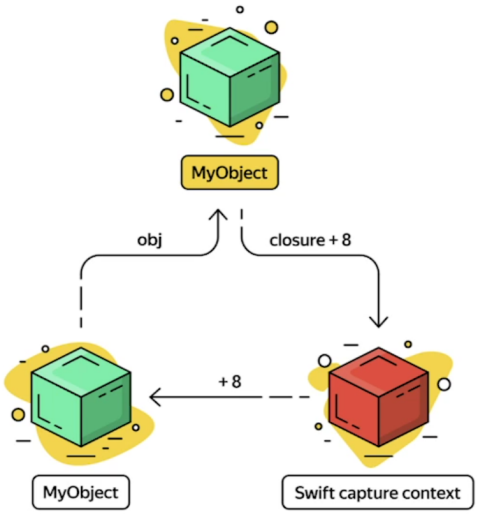
Таким образом осуществлялось ручное управление памятью в Objective-C до внедрения технологии автоматического подсчета ссылок (ARC - automatic reference counting). ARC является автоматическим в следующем понимании: расстановка вызовов функций alloc, retain, release, autorelease делаются компилятором за вас.

Важно понимать, что ARC не есть некий аналог garbage collector'a. ARC работает строго детерминировано, и на этапе компиляции. Компилятор, анализируя пути в вашем коде и понимая, где ваши объекты создаются, где границы владения этих объектов другим классами, и когда они должны быть уничтожены, расставляет на этапе компиляции нужные вызовы функций.

В рантайме у вас нет никакого дополнительного процесса, как garbage collector, который отслеживает память. Такой подход по сравнению с garbage collector дает ряд преимуществ. У вас нет остановок программы на сборку мусора, он (мусор), вприципе, не появляется. У вас память очищается тогда, когда это необходимо, а не в специально отведенные промежутки времени.

Однако эти преимущества даются определенной ценой. Для того, чтобы компилятор всегда правильно расставлял нужные вызовы функций, иногда ему требуется помощь. Ну и не забывать

про ряд определенных правил.
Так как ARC работает на базе счетчика ссылок, мы можем получить неприятную ситуацию, которая называется retain cycle.



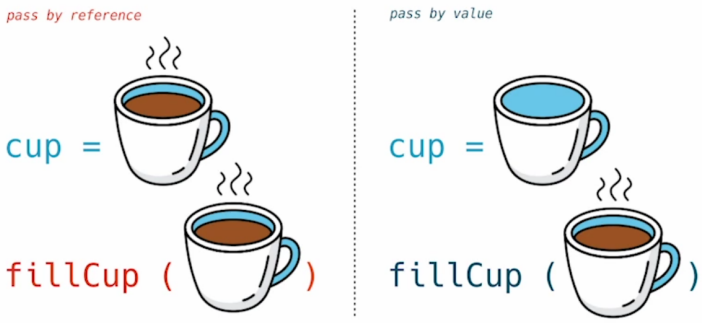
Суть ее в следующем. Когда у вас группа объектов по замкнутой цепочке ссылается друг на друга, и на них более никто не ссылается, то по идее эту группу объектов надо бы удалить. Однако ввиду того, что ни один из счетчиков ссылок не стал равным нулю, ARC не может их освободить. Так мы получаем утечку памяти. В памяти есть цикл, объекты которого ссылаются сами на себя, но на них более никто не ссылается.
Для того, чтобы этот цикл разорвать, и чтобы все заработало так, как надо, одну из ссылок в этом цикле необходимо пометить как weak. Weak, или висящие ссылки, не влияют на счетчик ссылок (и надо отметить, что являются optional). В таком случае все вызовы будут расставлены правильно и объекты будут удалены, утечки памяти не будет.
Одним из популярных случаев, когда у вас может случится retain cycle, являются отношения parent-child. Родитель имеет ссылку на дочерние элементы, а дочерние элементы имеют ссылку на родителя. В таких случаях ссылку на родителя рекомендуется делать weak-ссылкой. И тогда все будет работать хорошо.
Таким образом, программируя на swift, стоит задумываться в основном о првильной расстановке weak-ссылок. Однако из любых правил бывают исключения. И это исключение структуры, которое мы рассмотрим далее.

1.3.3. Структуры

Теперь давайте рассмотрим еще один вариант задания объектов через структуры. Объявить структуру в Swift можно вот так:

```
struct Car {
let mark: String
let model: String
}
```

Создать экземпляр этой структуры можно таким же образом, как мы создавали экземпляры классов, используя инициализатор:
print(Car(mark: "Bobik Motors", model: "Tuzik"))
В отличии от классов, если ваша структура имеет простую структуру (прошу прощения за тавталогию), писать инициализатор для нее не обязательно, компилятор автоматически сгенерирует стандартный инициализатор, в котором поля будут представлены в том же порядке, в каком вы их объявили, и тех же самых типов.
Если вам нужны какие-нибудь специальные инициализаторы, вы можете их задать вручную, но при этом вы теряете стандартный сгенерированный инициализатор, и его придется написать руками.
Теперь давайте рассмотрим отличие структур от классов. Для чего, если есть классы, понадобились структуры. Главное отличие структур от классов состоит в месте в памяти, где они хранятся. Здесь мы приходим к понятию ссылочных типов (reference-типов), и типов по-значению (value-типов). Такие виды типов существуют и в других языках программирования. Для тех кто с ними не знаком, поясню.
Существуют два возможных места в памяти, где экземпляры ваших типов могут располагаться – это стэк, и тогда типы объектов которые на нем располагаются, будут типами по значению, и куча, типы объектов которой будут соответственно ссылочными типами.
Разницу между ссылочными типами и типами по значению хорошо иллюстрирует следующая картинка:



Тип памяти, на которой хранится объект – это не единственное отличие структур от классов. Структуры не поддерживают наследование. Здесь вы можете использовать только агрегацию, если хотите делать какие-то сложные иерархии.
Для структур отсутствует приведение типов. И это логично, так как нет наследования. Вы не можете делать даункастинг (приведение к более частному типу), так и пользоваться базовым типом. Так как структуры являются типами по значению и находятся на стеке, для них не существуют, да и логично бесполезны деинициализаторы.
Структуры при передаче куда-нибудь всегда копируются, так как они находятся на стеке. Ввиду того, что типы по значению всегда копируются, не имеют смысла идеи множественного владения объектом, и поэтому для структур нет механизма подсчета ссылок. Если внутри структуры у вас есть изменяемое (var) поле, то при его изменении вы получите новый экземпляр структуры с измененным полем.
Благодаря тому, что структуры есть типы по значению, они всегда копируются, то они являются идеальным кандидатом для иммутабельных DTO (data transfer object) – объектов переноса данных. Сделав структуру иммутабельной, вы получаете потокобезопасность автоматически.
Ну и небольшая ремарка. Лучше, если нет на это особой надобности, всегда выставлять иммутабельный контракт (интерфейс), и делать его мутабельным, только если это действительно необходимо. Это убережет от ряда неочевидных ошибок и даст вам пространство для маневра при реализации вашего контракта.
Кроме этого, большинство стандартных объектов и контейнеров стандартной библиотеки, такие как массивы, словари и строки, являются структурами и, собственно, типами по значению.

1.3.4. ENUM'ы

ЕнУмы, они же перечисления. Swift, как и большинство популярных языков программирования, содержит механизмы для объявления перечислений, которые в Swift с их возможностями являются крайне полезным инструментом, особенно для написания понятных и самодокументированных API.
Простой enum объявляется с помощью ключевого слова enum. Давайте объявим перечисление планет:

```
enum Planets {
case mercury
case venus
case earth
case mars
case jupiter
case saturn
```

```
case uranus
case neptune
// NOTE: sadly, but no Pluto
}
```

Далее вы можете использовать Planets как имя типа, а элементы перечисления как его значения:

```
let p = Planets.mars
```

Важно отметить, что начиная со Swift 3, элементы перечисления объявляются с маленькой буквы. Это произошло в результате процесса Grand Renaming, о которой уже рассказывалось выше .

Простые enum'ы хоть и полезны, но не всегда применими, и поэтому Swift содержит ряд улучшений. Давайте их рассмотрим.

Для тех, кто работал с перечислениями в C++, наверняка помнят, что в плюсах каждому элементу перечисления ставится в соответствие некоторое значение типа int, идущее по порядку. В Swift по умолчанию такой возможности нет, никакому элементу перечисления никакое значение не ставится. Однако если вам это необходимо, то это можно сделать, и даже гибче, чем в C++. Для этого необходимо дополнительно объявить тип сопоставленного значения, который еще называют типом "сырого" значения (или raw value), и дальше сопоставить каждому элементу перечисления значение этого типа. Для нашего примера с планетами это будет выглядеть так:

```
enum Planets : Int {
  case mercury = 1
  case venus = 2
  case earth = 3
  case mars = 4
  case jupiter = 5
  case saturn = 6
  case uranus = 7
  case neptune = 8
}
```

И далее вы сможете получать доступ к этому "сырому" значению через поле rawValue:

```
print(Planets.mercury.rawValue)
```

В Swift можно не только ставить элементу перечисления в соответствие значение некоторого типа, но и иметь в элементе так называемые хранимые значения. Причем вы можете это делать не обязательно во всех элементах. Выглядит это так:

```
enum Result {
  case success(data: Data, headers: [String: Any])
  case failure(error: Error)
}
```

Такое использование перечислений полезно, например, при написании сетевого слоя, когда вам нужно вернуть результат выполнения запроса.
Следует помнить, что нельзя одновременно в рамках одного перечисления иметь и ”сырое” значение, и хранимые значения, поэтому либо ”сырое”, либо хранимое. Как быть если вам нужны хранимые значение, но при этом нужно некоторое сопоставленное значение – чуть позже.
Все элементы перечисления, так же как, например, в языке Java, являются почти полноценными объектами – вы можете объявлять внутри них вычислимые свойства и функции. Например, для нашего случая с планетами можно добавить вычислимое свойство, является ли планета планетой внутренней солнечной системы:

```
enum Planets {
  var isInnerPlanet: Bool {
    switch self {
      case .mercury, .venus, .earth, .mars: return true
      default: return false
    }
  }
}
```

И, например, функцию, которая вычисляет расстояние между планетами. Так как небесная динамика не является темой этой лекции, напишем эту функцию в виде заглушки на базе генератора случайных чисел:

```
enum Planets {
  func distance(to other: Planets) -> Double {
    return Double(arc4random())
  }
}
```

Теперь мы можем пользоваться этим свойством и функцией как обычно:

```
print(Planets.earth.isInnerSolarSystem)
print(Planets.mars.distance(to: .jupiter))
```

Я обещал показать как быть, когда нужны и хранимые значения, и сырые. Ввиду того, что элементы перечисления – это объекты, и у них можно объявлять вычислимые свойства – это

достаточно простая задача. Вернемся к нашему примеру с результатом выполнения сетевого запроса:

```
enum Result {
  case success(data: Data, headers: [String: Any])
  case failure(error: Error)
}
```

Мы хотим иметь для элементов перечисления сырые значения строкового типа, это легко сделать так:

```
enum Result {
  var rawValue: String {
    switch self {
      case .success: return "success"
      case .failure: return "failure"
    }
  }
}
```

Также, чтобы получить в свое использование хранимые значения элемента перечисления, необходимо также воспользоваться опереатором switch:

```
let res: Result = ...
switch res {
  case let .success(data, headers): print(data, " ", headers)
  case let .failure(error): print(error)
}
```

На этом с перечислениями пока все. Мы ещё вернемся к ним в дальнейшем. А теперь перейдем к рассмотрению написанию обощенного кода с помощью Generics.

1.3.5. Generics

В завершении рассмотрения возможностей объектно-ориентированного программирования в Swift давайте рассмотрим generic'и. Generic'и – это инструмент для написания обобщенного кода. Типы, используемые при объявлении generic'ов, можно рассматривать как своего рода placeholder'ы, в которые пользователь вашего кода будет подставлять свои типы.
Давайте рассмотрим простой пример. Мы хотим написать функцию сложения двух объектов. В самом простом случае мы хотим складывать два целых числа:

```
func sum(lhs: Int, rhs: Int) -> Int {
  return lhs + rhs
}
```

Теперь усложним задачу. Мы хотим написать еще и функцию сложения двух чисел с плавающей точкой, и нам потребуется написать еще одну реализацию:

```
func sum(lhs: Double, rhs: Double) -> Double {
  return lhs + rhs
}
```

Потом мы захотим складывать две даты, две валюты и т.д. И каждый раз нам придется написать эту функцию снова еще раз... Этого очень хотелось бы избежать, и нам на помощью приходят generic'и. Нам необходимо объявить, что наша функция будет использовать некий тип, пока без ограничений, будет принимать два параметра этого типа, и возвращать результат этого же типа. Какой конкретно тип – нам не важно. Справедливости ради стоит сказать, что вообще-то важно, мы хотим чтобы существовал опереатор сложения для такого типа, но это пока опустим.
Поэтому перепишем нашу функцию с учетом обобщенного типа. Сам код практически не поменяется, но теперь реализацию мы пишем один раз, а используем несколько. Итак, функция:

```
func sum<T>(lhs: T, rhs: T) -> T {
  return lhs + rhs
}
```

И использовать ее можно так для всех типов, которые поддерживаются оператором сложения:

```
print(sum(lhs: 1, rhs: 2)) // 3
print(sum(lhs: 1.5, rhs: 3.5)) // 5.0
```

Однако иметь только обобщенные функции недостаточно. Хотелось бы иметь возможность объявлять и использовать обобщенные типы. Например, это очень необходимо для контейнеров. И такая возможность есть. Причем мы можем объявлять как обобщенный класс, так и обобщенную структуру.
Давайте рассмотрим на примере, мы хотим объявить контейнер "упаковка", в которую мы сможем складывать и забирать объекты. Порядок нам не важен:

```
struct Pack<Obj> {
}
```

Так как реализацию работы с сырой памятью писать не хочется, воспользуемся стандартным массивом для хранения объектов:

```
struct Pack<Obj> {
   var storage: [Obj] = []
}
```

Как можно заметить, в качестве типа мы используем Obj, пока не накладывая никаких ограничений, кроме тех, которые нужны для того, чтобы сформировать массив такого типа.
Еще нам необходима функция размещения объекта в упаковку:

```
  mutating func pack(obj: Obj) {
    storage.append(obj)
  }
```

И функцию, изымающую объект из упаковки. Условимся, что берем первый объект, и помним, что это не очень эффективно в части работы с памятью, но нам для примера достаточно. Обратите внимание на Optional. Не всегда мы сможем извлечь объект, так как упаковка может быть пуста:

```
  mutaing func unpack() -> Obj? {
    guard !storage.isEmpty else { return nil }
    return storage.remove(at: 0)
  }
```

Использовать нашу упаковку мы будем на примере строк:

```
var pack = Pack<String>()
```

Положим туда дрель и сверла:

```
pack.pack("Drill")
pack.pack("Bolt")
pack.pack("Bolt")
```

И потом будем потихоньку доставать объекты оттуда:

```
print(pack.unpack()) // Optional<Drill>
print(pack.unpack()) // Optional<Bolt>
print(pack.unpack()) // Optional<Bolt>
print(pack.unpack()) // nil
```

Сначала мы достанем дрель, потом сверла, потом объекты закончаться и доставать будет нечего, наш контейнер начнет возвращать nil.
Иногда на тип, который мы используем, необходимо наложить некоторые ограничения. Это можно сделать, например, через протокол. Давайте объявим какой-нибудь протокол, для объектов

которые можно упаковать:

```
protocol Packable {}
```

И далее потребуем, чтобы тип объектов, которые мы размещаем в нашей упаковки, был Packable:

```
struct Pack<Obj: Packable> { ... }
```

Код со строками ожидаемо оказался сломан, так как строка не поддерживает протокол Packable. К счастью, это легко исправить с помошью расширений:

```
extension String: Packable {}
```

И наш код про дрель и сверла снова начнет работать.
А что, если мы хотим объявить обобщенный протокол? Увы, обобщенный протокол в таком виде в Swift объявить нельзя. Однако для протоколов существует другая технология, которая называется "протоколы с ассоциированными типами":

```
protocol PackingContainer {
  associatedtype Element
  mutating func pack(obj: Element)
  mutating func unpack() -> Element?}
```

Что говорит о том, что этот протокол будет оперировать с неким типом. Для того, чтобы реализовать этот протокол, вы явно с помощью typealias должны объявить тип элемента:

```
struct IntPack : PackingContainer {
  typealias Element = Int
}
```

Или реализовать требуемые функции протокола с каким-то определенным типом, и компилятор попробует вывести его самостоятельно.

1.4. Протокол-ориентированное программирование

В предыдущем видео про generic'и мы уже затрагивали понятия протокола и ассоциированного типа для этого протокола, теперь давайте рассмотрим протоколы более подробно. Вообще, в Swift протокол – это сущность, которая позволяет описать интерфейс или контракт некого объекта, но при этом не содержит деталей реализации этого контракта. Если вы знакомы с языками Java или C, том там эта сущность выражена понятием interface.
Следовательно, контракт – это то, что дает пользователю понимание по составу полей и функций объекта, но не содержит даже намека на то, как он конкретно реализован.

В Swift протокол можно объявить следующим образом:

```
protocol Vehicle {
var isMoving: Bool { get }
var name: String { get set }
func move(to pos: CGPoint) -> Bool
}
```

В данном случае любой, кто реализует этот протокол, должен иметь поле на чтение isMoving, поле на чтение и запись name, а также содержать функцию move(to:). Давайте реализуем такой класс. Объявление поддержки протокола похоже на объявление наследования. Итак, класс автомобиль, который является машиной:

```
class Car : Vehicle {
  var isMoving: Bool = true
  var name: String = "Tuzik"
  func move(to: CGPoint) {
  }
}
```

Поля можно реализовывать как хранимыми полями, так и вычислимыми, разницы с точки зрения пользователя протокола между ними нет.
Важно помнить, что в Swift наследование одинарное. Класс может иметь только один суперкласс, здесь нет множественного наследования как в C++. Однако протоколов вы можете поддерживать столько, сколько захотите. Это позволяет реализовать механизм граней (traits) на базе протоколов и расширений к ним.
Как я уже говорил в лекции про generic'и, протоколы могут иметь некоторый ассоциированный тип, и, таким образом, вы можете сделать своего рода generic protocol'ы:

```
protocol Soldier {
  associatedtype Weapon
  var weapon: Weapon { get }
}
```

Класс, который будет поддерживать такой протокол, должен назвать тип, либо реализовать контракт с помощью какого-то конкретного типа:

```
class Trooper : Soldier {
  var weapon: Machinegun = Machinegun()
}
```

Либо явно определив его через typealias:

```
class Sniper : Soldier {
  typealias Weapon = SnipperRiffle()
}
```

Одна из наиболее интересных особенностей Swift – это наличие расширений. С расширениями на протяжении лекций мы уже немножко сталкивались, но теперь давайте рассмотрим их подробнее. Расширения позволяют вам добавить некоторые свои вычислимые, и только вычислимые (это важно), поля и функции к какому-нибудь классу, причем не обязательно вашему, причем не обязательно иметь исходники этого класса. Важно, что вы должны делать это на базе публично доступного интерфейса этого класса. Поэтому и не важно иметь исходники.
Например, мы можем для класса String из стандартной библиотеки реализовать функцию md5:

```
extension String {
  func md5() -> String {
    /return...
  }
}
```

К слову, расширения штука не новая, это уже было в Objective-C, и там называлось категориями. Тем, кто знаком с Objective-C, должно быть знакомо.
Расширения могут быть публичными, и соответственно, в зависимости от спецификаторов доступны в рамках модуля, или даже вовне, и бывают расширения закрытые, доступные только в данной единице трансляции (то есть один swift'овый файл). Часто закрытые расширения полезны для выноса – переиспользования некой части кода, или для улучшения читаемости вашего кода.
Вспомним наше перечисления планет Planets. Некому пользователю потребовалось знать диаметр планеты для каких-то расчетов. Вместо того, чтобы каждый раз в необходимых местах switch по этому перечислению, этот пользователь может реализовать закрытое расширение таким образом:

```
private extension Planets {
  var diamter: Double {
    switch self {
      case .mercury: return 10
      ...
    }
  }
}
```

Если это расширение нигде за рамками данного файла не нужно, его можно объявить как private,

и тем самым сэкономить в размере бинарника и времени компиляции. Иногда такие расширения и не должны быть видимыми за пределами этого файла.
Ну, и как я уже говорил в ролике про generic'и, протоколы можно использовать в ограничениях на тип generic'а. Напомню, как это выглядело в примере с контейнером Pack и ограниченим с протоколом Packable:

```
struct Pack<Obj: Packable> { ... }
```

И если вы хотите поддержать соответствие протоколу внешнему классу, то вы легко это можете сделать с помощью расширения, как это мы сделали со строкой:

```
extension String : Packable { ... }
```

Настало время рассмотреть стандартную билиотеку, и начнем мы с массивов и словарей.

1.5. Массивы и словари

Теперь давайте перейдем к рассмотрению двух основных классов стандартной библиотеки. Первая структура данных, с которой мы сталкиваемся в любой языке программирования – это массив. В Swift массивы очень тесно интегрированы в язык, вы можете использовать их большим количеством разных способов. Массив, например, целых чисел, можно объявить просто перечислив его элементы:

```
let array1 = [1, 2, 3]
```

Или создать его через инициализатор с небольшим добавлением синтаксического сахара таким образом:

```
let array2 = [Int]()
```

На самом деле мы здесь неявно (через синтаксический сахар) пользуемся структурой (да в Swift массив - это структура) Array, поэтому объявить массив можно и так:

```
let array3 = Array<Int>()
```

Массив в Swift – это Generic структура, поэтому вы можете использовать его с любым нужным вам типом. Если вы хотите управлять выводом типов, и хотите задать тип элемента явно, это полезно, когда вы объявляете массив через базовый класс или даже протокол, то можно добавить спецификатор типа:

```
let array4: [Int] = [Int]()
let array4: [Int] = [1, 2, 3]
```

Но если контекста достаточн, и компилятор может вывести тип самостоятельно, то в этом нет необходимости.
Массивы в Swift позволяют обращаться к элементу массива по индексу также, как это принято во многих языках, через так называемые индексеры:

```
print(arr1[1])
```

Нумерация элементов массива начинается с нуля.
Приведенные выше примеры объявляют константные массивы, вы не можете их изменять. Для того, чтобы объявить изменяемый массив, достаточно ключевое слово let заменить на ключевое слово var. Дальнейший синтаксис объявления выглядит точно также. Однако, объявив массив изменяемым, вы получаете возможность пользоваться функцией append:

```
var xarr = [1,2,3]
xarr.append(5)
```

которая добавляет новый элемент в конец массива. Или, например, функцией insert:

```
xarr.insert(6, at: 3)
```

которая позволяет вам добавить элемент по какому-нибудь индексу. Добавление по несуществующему индексу вызовет исключение. Также вы можете удалить элемент по индексу с помощью функции remove:

```
xarr.remove(at: 0)
```

Или удалить вообще все элементы из массива:

```
xarr.removeAll()
```

Как проитерироваться по массиву? Выше, когда я рассказывал про цикл for я отмечал, что в Swift нет привычного по C/C++ вида цикла for, но есть встроенный цикл foreach, а массив в Swift является итерируемым объектом, поэтому по элементам массива можно итерироваться так:

```
for element in array {
  ...
}
```

Если в ходе итерации вам потребуется еще и индекс элемента, то это можно сделать с помощью специальной функции enumerated, которая в ходе итерации возвращает не элементы массива, а кортеж индекс-элемент:

```
for (index, element) in array.enumerated {
}
```

Можно еще итерироваться с помощью функции forEach, но о ней позже, когда будем говорить про замыкания.
Во многих языках такая структура данных как словарь, или map, не является встроенной в язык, однако Swift является приятным исключением. В Swift кроме массивов в язык встроены и словари ключ-значение. Как и в случае с массивам есть стандартная структура Dictionary (да и словарь тоже value type), но также существует огромное количество разного синтаксического сахара, которое вы можете использовать для объявления словарей.
Стоит, пожалуй, отметить, что не любой объект может быть ключом словаря. Для этого объекту необходимо реализовать протокол AnyHashable, то есть иметь оператор сравнения и функцию вычисления хэша. Простой словарь с ключами типа Int и значениями типа String можно объявить просто перечислив варианты так:

```
let dict = [1: "One", 2: "Two"]
```

Или через инициализатор с применением сахара:

```
let dict = [Int: String]()
```

Или, собственно, через структуру Dictionary:

```
let dict = Dictionary<Int, String>()
```

Также, как и в случае с массивами, если вы хотите явно задать типы ключа и значения, вы можете добавить спецификатор типов:

```
let dict1: [Int: String] = [1: "One", 2: "Two"]
let dict2: [Int: String] = [Int: String]()
```

Вы можете обращаться к элементами словаря, также как и в массивах с помощью индексера. Единственное и очень важное отличие состоит в том, в отличии от массива, который генерирует исключение при попытке обратится по несуществующему индексу, словари вместо этого возвращают вам optional. Если элемент по такому ключу есть, вы получите заполненный этим элементом optional, если же значения по этому ключу отсутствует, вы получите nil:

```
print(dict[1])
```

Также как и в случае с массивами, вы можете создавать изменяемые словари с помощью ключевого слова var. После этого вы получаете возможность добавлять элементы в словарь с помощью индексеров:

```
var xdict = [Int:String]()
xdict[1]="One"
```

И с помощью этого же индексера вы можете удалять элементы из словаря просто присвоив nil:

```
xdict[1] = nil
```

Также как и в массивах, словари в Swift позволяют итерироваться по ним с помощью цикла foreach. Здесь все просто – итератор словаря возвращает сразу кортеж ключ-значение:

```
for (key, value) in dict {
}
```

Порядок элементов, строго говоря, не задан.
Мы рассмотрели уже много про объектно-ориентрованное прога=раммирование, однако swift – мультипарадигменный язык, и содержит элементы функционального программирования, которое мы рассмотрим далее.

1.6. Функционально-ориентированное программирование

1.6.1. Функционально-ориентированное программирование

Теперь давайте рассмотри еще один важный элемент языка Swift - замыкания или closures, иногда их еще называют лямбдами. Замыкание можно рассматривать как некую анонимную функцию. В использовании она крайне проста, так мы можем объявить замыкание, которое ничего не принимает, ничего не возвращает и ничего извне не захватывает:

```
let sayHello = {
  print("Hi")
}
```

Объвленное имя переменной или константы становится именем, с помощью которого замыкание можно вызвать:

```
sayHello()
```

Как уже было сказано ранее, функция, которая ничего не принимает и ничего не возвращает, имеет весьма ограниченную область применения. Поэтому нужен способ объявить передаваемые внутрь замыкания параметры.
Если вы объявляете замыкание прямо на месте, у вас нет предобъявленного типа, которому можно соответствовать, тогда вам ничего не остается, кроме как использовать полный синтаксис объявления параметров:

```
let greetings = { (_ p: String) ->Void in
  print("Hi, \(p)")
}
```

Однако если у вас есть преодпределенный тип, или можно воспользоваться указанием типа при объявлении, то тогда можно использовать сокращенный синтаксис:

```
let greetings: (String) -> Void = { name in
  print("Hello, \(name)")
}
```

Если вы хотите из замыкания что-нибудь вернуть, то в случае предопредленного типа ничего делать более не нужно, достаточно вернуть значение нужного типа. Объявить возвращаемое значение при условии отсутствия предопределенного типа можно так:

```
let rnd = { () -> String in
  return UUID().uuidString
}
```

Пустые скобки означают, что это замыкание не принимает никаких входных параметров. Как известно, замыкания могут захватывать внутрь себя некоторые константы и переменные извне, то есть которые объявлены и используются вне тела замыкания. Существует ряд правил, которые регламентируют такой захват. Если тип захвата явным образом не объявлен и просто используете внешнюю переменную внутри тела замыкания, то эта переменная будет захвачена по сильной ссылке, например, так:

```
var printer = Printer()
let sayHello = {
  printer.print("Hi")
}
```

Однако если вы хотите захватить переменную по значению, например, структуру, или вы явно хотите иметь внутри замыкания именно это значение, то это надо явно объявить:

```
let sayHello = { [printer] in
  printer.print("Hi")
}
```

Иногда необходимо осуществить захват по слабой ссылке, например, когда замыкание сохраняется как поле, и внутрь надо захватить ссылку на сам объект для разрыва retain cycle'a:

```
let sayHello = { [weak printer] in
  printer?.print("Hi")
}
```

Важно помнить, что под таким именем внутри тела замыкания переменная будет обернута в optional, и его надо правильно отрабатывать (разыменовывать или хотя-бы использовать optional

chaining). Может так случится, что значения в этом optional на момент вызова замыкания может и не быть.
Если вы не хотите иметь optional и вы точно знаете, что этот объект проживет достаточно долго, чтобы имелась возможность его использовать внутри замыкания, вы можете объявить захват как unowned:

```
let sayHello = { [unowned printer] in
  printer.print("Hi")
}
```

При этом стоит иметь ввиду, что в части подсчета ссылок unowned работает как weak, но не оборачивает значение в optional. Поэтому если объект будет все-таки удален раньше, чем его использует замыкание, вы получите runtime crash при попытке доступа к этому объекту. Поэтому unowned стоит применять, только когда вы ясно понимаете, что вы делаете, и когда у вас есть железобетонные гарантии того, что объект будет жить достаточно. Чаще всего все же предпочительнее использовать weak.
Тут стоит сделать некоторое отступление и рассмотреть такое понятие как убегающие (escaping) замыкания. Существуют понятие убегающих (escaping) и неубегающих (non-escaping) замыканий. Если явным образом ничего не объявлять, то по умолчанию в Swift замыкания неубегающие, они должны быть вызваны, и должны вернуть управление в рамках scope'а, в котором объявлены. Иногда же необходимо сохранять замыкание как поле класса и выполнять асинхронный вызов, в таком случае вам понадобятся escaping замыкания. От non-escaping они отличаются тем, что могут быть вызваны вне scope'а, в котором объявлены, и здесь стоит помнить про ARC и управление памятью, и могут быть вызваны на другом потоке, а тут стоит помнить про гонки, дедлоки, и другие тонкости многопточности. Убегающие замыкания часто используются для так называемых completion handler'ов – функций обратного вызова (callback'ов).
Я уже говорил про захват внешних переменных по сильным и слабым ссылкам. хочу еще раз остановится на таком захвате, но уже на захвате self. Если внутри класса вы используете self в замыкании, то вы очень легко можете получить retain cycle, например, сохранив экземпляр замыкани как поле класса. Для того, чтобы этого избежать, стоит захватывать self внутри замыкания по слабой ссылке и проверять optional. Компилятор при явном использовании self внутри замыкания вам про retain cycle не подскажет.
А теперь давайте рассмотрим что из функционального программирования есть в стандартной библиотеке.

1.6.2. Функциональные элементы стандартной библиотеки

В прошлом видео мы рассматривали имеющиеся в Swift элементы функционально-оринетированного программирования. Сейчас же давайте рассмотрим специальные функциональные элементы,

имеющиеся в стандартной библиотеке языка Swift.
Первое, что мы рассмотрим – это функция map. Функция map позволяет вам в функциональном виде задать некую трансформацию коллекции. Например, у нас есть массив целых чисел, и мы хотим к каждому из них добавить единицу. Мы можем написать традиционный цикл for. В таком случае нам потребуется изменяемый массив, в котором мы произведем поэлементное сложение. С помощью функции map мы можем получить этот массив как результат вызова функции с уже измененными элементами, нет необходимости явно управлять этим изменяемым массивом.
Следовательно, у нас есть массив целых чисел arr:

```
let array = [1, 2, 3, 4]
```

Целевая функция map будет выглядеть так, она принимает замыкание, с помощью которого будут поочередно трансформироваться элементы:

```
let res = array.map { $0 + 1 }
```

Тут надо отметить, что писать полную версию замыкания не всегда есть необходимость, можно пользоваться вот такой сокращенной формой, где элементы замыкания нумеруются таким вот образом.
Функцию map мы также можем использовать и для словарей, однако для словарей имеется одно очень важное отличие – map для словарей возвращает массив, и внутри замыкания обратывается не элемент, а кортеж ключ-значение. Также функция map может применяться для optional. В таком случае значение optional, если оно есть, трактуется как единственный элемент коллекции.
Существует также принятая во многих других функциональных языках функция flatMap (одна из ее перегрузок), которая из вложенных массивов делает плоский:

```
let arr = [[1, 2], [3, 4]]
let flat = arr.flatMap { $0 }
```

Однако самое интересное не в этой перегрузке. И дело вот в чем. Иногда в ходе трансформации вы можете получить результирующий тип optional, и в ходе трансформации могут получаться значения nil. Вы можете захотеть получить результирующую коллекцию очищенной от значений nil. В этом нам поможет функция flatMap, или в новой редакции Swift 4.1 для массивов – compactMap, для остальных коллекций и optional – flatMap. Так или иначе, как бы они не назывались, делают они одно и тоже. Они работают примерно также как, и map, но со своеобразной встроенной проверкой на nil. Если в ходе трансформации получено значение, то оно будет возвращено, если же получен nil – он будет исключен из итогового результата. Например, у нас есть optional массив вот такого вида:

```
let arr = [1, 2, nil, 3, nil, 4, nil, 5]
```

Тогда воспользовавшись функцией compactMap, мы можем получить его очищенным так:

```
let res = arr.compactMap { $0 }
```

Еще раз напомню, что flatMap можно применять и для optional.
Функции map/flatMap/compactMap могут трансформировать коллекции поэлементно, однако иногда возникает необходимость по коллекциям посчитать какие-либо агрегаты, и здесь нам на помощь приходит функция reduce. Она принимает два значения. Первое - это исходное значение целевого агрегата, второе - замыкание, которое в свою очередь принимает текущее значение агрегата и очередной элемент. Например, мы хотим просуммировать элементы массива целых чисел:

```
let arr = [1, 2, 3]
```

Мы легко можем сделать это с помощью функции reduce:

```
let sum = arr.reduce(into: 0) { res, item in
  res += item
}
```

Как я уже выше говорил, функция map для словарей возвращает массив, однако иногда требуется трансформировать массив, или словарь в словарь, и с помощью map это сделать нельзя. Однако с помощью reduce можно. Например, у меня по прежнему есть массив целых чисел:

```
let arr = [1, 2, 3, 4]
```

И я хочу получить на выходе словарь, в котором ключом будет элемент, а значением его строковое представление. С помощью reduce это можно сделать так

```
let dict = arr.reduce(into: [Into:String]()) { res, item in
  res[item] = " \ (item)"
}
```

С помощью flatMap/compactMap вы можете делать определенного рода фильтрацию коллекций, просто возвращая nil для элементов которые вам не нужны, однако для фильтрации коллекций есть специальная функция filter, и лучше пользоваться ей во всех случаях, когда вам нужно именно отфильтровать коллекцию а не исключить nil'ы. Функция filter принимает замыкание, в которое передается каждый элемент, а на выходе вы должны вернуть булево значение, которое показывает, хотите вы иметь этот элемент в результате, или нет. Это позволяет вам весьма гибко задавать правила фильтрации словарей:

```
let res = arr.filter { $0 > 2 }
```

Часто вам нужно найти индекс какого-либо элемента в колекции, это можно сделать с помощью функции index(of:), важно чтобы тип элемента поддерживал Equatable (специальный протокол,

объявляющий что у типа есть оператор проверки на равенство). Например, мы хотим найти индекс элемент 3 в нашем массиве целых чисел:

```
let index = arr.index(of: 3)
```

Бывают случаи, когда конкретное значение вы не знаете, вам нужен первый элемент, удовлетворяющий какому-то конкретному условию. У вас есть некий предикат – то самое условие. Для этого вы можете воспользоваться функцией first(when:) и передать в нее этот предикт. Например, я хочу найти элемент, который больше двух. Важно помнить, что элемента удовлетворяющего предикату в коллекции может и не быть, поэтому эта функция возвращает optional, который надо соответствующим образом обрабатывать:

```
let element = arr.first { $0 > 2 }
```

Когда речь шла о циклах, я говорил о том, что есть специальная функциональная конструкция, которая также позволяет итерироваться по коллекциям – forEach. Она позволяет вам пройтись по каждому элементу коллекции с помощью замыкания и что-то сделать:

```
arr.forEach {
  print($0)
}
```

Для словарей ситуация похожая на map – в качестве значения замыкания будет передан кортеж ключ-значение.
Мы рассматрели последовательно процедурную, объектно-ориентированную, и вот теперь функциональную сторону языка Swift. Мы уже движемся к концу, и теперь давайте рассмотрим разного рода полезности, которые в том числе понадобяться вам при выполнении домашнего задания. Начнем с сопоставления с образцом и конструкции switch.

1.7. И другие полезности

1.7.1. Switch и сопоставление с образцом (Pattern matching)

В заключение видеолекции про язык Swift хочется поговорить про разного рода полезности. Первой из них будет очень мощная по возможностям конструкция switch. В Swift она не просто позволяет вам проверить значение переменной по некоторому набору значений, а позволяет делать так называемое сопоставление с образцом, или более привычное из английского языка – pattern matching.
Вообще, switch в Swift выглядит примерно также как, и в большинстве других языков:

```
switch a {
  case 1: print("1")
  case 2: fallthrough
  case 3: print("More than one")
  default: break
}
```

Правда есть одна небольшая, но очень важная разница – в swift не требуется явное написание break. Если вы его явно не напишите, у вас не будет автоматической обработки следующих case'ов, то есть если мы уже явно попали в какой-то из case'ов, то другие обрабатываться не будут. Если же вы явно хотите традицонную, например, для C++ логику с "проваливанием", то для этого надо специально объявить с помощью fallthrough, которая позволит вам это сделать. Правда, в приведенном примере можно было бы просто разделить варианты 2 и 3 запятой. Впрочем, конструкция break все же иногда нужна, например, когда вы хотите явно заявить какой-то case, но обрабатывать в нем нечего. По соглашению, в конструкции case должно быть хотя бы одно выражение, и если вам нечего написать, можно заменить на break.
Однако давайте вернемся к обещанной мощи конструкции switch в Swift. Самое первое, что стоит упомянуть – возможность сопоставлять кортежи. Например, у нас есть два булевский значения:

```
let a = true
let b = false
```

и мы хотим сделать некую логику в зависимости от их совокупных значений. С помощью switch это можно сделать достаточно просто:

```
switch (a, b) {
  case (false, false): print("0")
  case (false, true): print("1")
  case (true, false): print("2")
  case (true, true): print("3")
}
```

Двоичная логика! :)
В конструкциях case можно использовать маски, или wildcards, когда вы не хотите явным образом задавать значение или переменную. Например, у нас есть некотороя optional строка:

```
let p: String? = ...
```

С помощью конструкции switch и масок можно в зависимости от содержимого optional реализовать какую-то логику

```
switch p {
  case _?: print("Value!")
  case nil: print("No Value!")
}
```

Справедливости ради, optional значения можно поматчить и по-другому. Так как optional – это перечисление, через элементы перечисления:

```
switch p {
  case .some: print("Value!")
  case .none: print("No Value!")
}
```

Возвращаясь к маскам, их можно использовать не только для хитрой проверки optional, но и, как я уже сказал – для опускания тех значений, которые явным образом вам не нужны. Например, у нас есть вот такой хитрый кортеж:

```
let val = (5, "example", 3.14)
```

И мы хотим получить из него только значение числа pi:

```
switch val {
  case (_, _, let pi): print("pi: \(pi)")
}
```

Может случиться так, что внутри конструкции case вам потребуется значение из условия switch, или по кортежу, или даже можно сделать деконструкцию кортежа на переменные, тогда можно воспользоваться конструкцией case let:

```
switch (4, 5) {
  case let (x, y): print("We are at \(x) \(y)")
}
```

Сопоставление с образцом на кортежах можно довести до невиданных высот. Например, у нас есть ряд переменных – целочисленная optional строка и некий объект:

```
let age = 13
let job: String? = "Worker"
let userInfo: AnyObject = NSDictionary()
```

И мы хотим забиндить возраст, проверить, что есть должность, и что есть некий объект – это NSDictionary. С помощью switch это можно сделать:

```
switch (age, job, userInfo) {
  case (let age,_?, _ as NSDictionary): print("Hurray")
}
```

Биндинг значений особенно хорошо себя показывает при работе с перечислениями, в которых есть ассоциированные значения. Например, у нас есть вот такое перечисление, которое описывает сущности в некоторой компьютерной игре. У каждой сущености есть некоторые координаты:

```
enum Enitites {
  case soldier(x: Int, y: Int)
  case player(x: Int, y: Int)
  case tank(x: Int, y: Int)
}
```

У нас есть массив этих сущностей:

```
let entities: [Entities] = [.tank(x: 1, y: 1), .player(x: 2, y: 3)]
```

и хотим что-то с ними сделать. При этом нам нужны координаты:

```
for e in entities {
  switch e {
    case let .soldier(x, y): print("S at \(x) \(y)")
    case .player(let x, let y): print("P at \(x) \(y)")
  }
}
```

Обратите ваше внимание на то, что let можно писать как в после case, так и перед желаемым именем переменной. И такая, и другая конструкция возможны, но когда переменных много, let придется писать много раз, что не так красиво, как при использовании case let.
Конструкция switch в Swift позволяет делать сопоставление и одновременно downcasting. Вы можете проверить на нужные тип, или даже получить переменную нужного типа. Например, у нас есть некая переменная a:

```
let a: Any = 5
```

Если мы хотим сделать что-то, если там значение типа Int:

```
switch a {
  case is Int: print("Hurray, Int!")
}
```

Или же мы даже можем получить это значение в виде переменной типа Int:

```
switch a {
  case let n as Int: print("Int: \(n)")
}
```

С помощью switch можно проверять попадение значений в диапазоны. Например, проверить, что некоторое целое число попадает в диапазон:

```
switch 5 {
  case 0...10: print("0-10")
}
```

Если же всего вышеперечисленного вам недостаточно, и у вас есть своя хитрая логика, нужная вам для каких-то ваших задач, вы можете переопределить оператор =, который будет сравнивать с наличием жизней у некого солдата:

```
struct Soldier {
  let hp: Int
  let x: Int
  let y: Int
}

func ~= (pattern: Int, value: Soldier) -> Bool {
  return pattern = value.hp
}
```

и затем написать конструкцию switch, в которой этот оператор будет использоваться:

```
switch soldier {
  case 0: print("Dead soldier")
}
```

Ну и дабы вы до конца прониклись всей мощью конструкции switch, давайте напишем функцию вычисления чисел фибоначчи, что называется Like a Boss:

```
func fibonacci(\_ i: Int) -> Int {
  switch(i) {
    case let n where n <= 0: return 0
    case 0, 1: return 1
    case let n: return fibonacci(n - 1) + fibonacci(n - 2)
  }
}
```

Switch в swift, как вы могли убедиться, представляет большой спектр возможностей. Используйте его для написания элегантного кода. А теперь давайте вернемся к общим вещам. Я уже немножко рассказывал про процесс великого переименования, давайте остановимся на нем подробнее.

1.7.2. Соглашение по именованию и переименованию

В этом видео давайте поговорим не столько про то, как что-то записать и оформить в Swift, сколько про то, как именовать разного рода сущности.
Если вам доводилось работать с Objective-C, то информация следующих пары минут вам покажется знакомой, если же не доводилось, то это небольшое введение для вас.
Objective-C отличается от большинства известных объектно-ориентированных языков своим способом вызова методов класса – здесь это не столько вызов методов, сколько так называемая посылка сообщений. Например, если у нас есть некий класс Person, и нам надо создать объект этого класса, то мы классу посылаем сообщений alloc:

```
[Person alloc]
```

для выделения памяти, достаточного для размещения этого объекта. Сама функция alloc занимается только тем, что выделяет объем памяти, после этого у нас есть указатель на эту память, который типизирован как объект класса, однако пока там мусор. Для того, чтобы эта память была корректно инициализирована, нам необходимо послать этому адресу сообщение инициализации:

```
[[Person alloc] initWithName: "Vasya Pupkin" andMoney: 100500]
```

Здесь мы создаем персону ”Вася Пупкин”, у которого есть 100500 денег. Дальше мы хотим, чтобы этот человек сходил в магазин и купил айфон (денег у него 100500 - на айфон хватит). Поэтому мы посылаем ему сообщение ”пойди в магазин и купи айфон”:

```
[person goTo: appStore andBuy: iPhone]
```

Таким образом, в objective-c принято именовать все методы так, чтобы в момент вызова выражение отправки сообщения какому-нибудь объекту выглядело как правильная фраза на английском языке. В случае с походом в магазин это выглядит как предложение: человек – подлежащее, ”пойди в магазин и купи айфон” выглядит как фраза на английском языке.
Поэтому в objective-c принято такое многословное наименование методов для того, чтобы они в месте вызова выглядели как фразы на английском языке.
После того, как Apple выложила Swift в Open Source, тогда это был Swift 2, все методы из objective-c в Swift мапились практически 1 в 1 как есть. Однако быстро стало понятно, что в Swift, в котором способ вызова функций больше поход на традиционные объектно-ориентированные

языки, становится слишком много ненужных слов, и не выполняется одно из основных положений, которые лежали в основе Swift – это краткость, и в то же время емкость того, что написано. Код создания персоны в Swift 2 выглядел бы так:

```
let person = Person(withName: "Vasya Pupkin", andMoney: 100500)
```

В процессе перехода к Swift 3 произошел процесс, который получил название Grand Renaming, или великое переименование. В ходе которого имена функции, которые мапяться из objective-c в Swift, подверглись большому усечению. Все лишние с точки зрения swift слова были выкинуты. В Swift 3 и сейчас создание экземпляра класса человек выглядит так:

```
let person = Person(name: "Vasya Pupkin", money: 100500)
```

Ну и, соотвественно, поход в магазин за покупкой айфона будет выглядеть так:

```
person.go(to: shop, buy: iphone)
```

В ходе Великого Переименования, и в ходе ряда следующих релизов языка, традиционные в objective-c сущности изменили даже свой тип. Например, функции GCD:

```
dispatch_async(dispatch_main_queue(), ^{
  dispatch_async(dispatch_get_global_queue(), ^{
  })
})
```

в objective-c и в первых версиях Swift были именно функциями, которые работали как в старом добром C – функция, которая первым аргументом принимает структуру, и далее что-то с ней делает. Однако уже в третьем Swift функции для работы с GCD превратились в классы и их методы, например, DispatchQueue, DispatchSemaphore и так далее. Сейчас приведенный код GCD выглядит так:

```
DispatchQueue.main.async {
  DispatchQueue.global().async {
  }
}
```

Подобному преобразованию подверглись сущности, которые всегда были структурами, и которые имели ряд глобальных функций для работы с ними. Например, структуры из библиотеки Core Graphics – CGRect и CGPoint. Теперь они стали структурами Swift; функции CGPointMake и CGRectMake стали инициализаторами этих структур, с размеченными аргументами.
Также при должной разметке из обычных C-шных перечислений можно получать перечисления Swift, а не просто глобальные константы, как это было в предыдущих релизах. Это перечисления с сырым значением (raw value).

Таким образом, именование объектов в Swift – это вопрос лаконичности. Однако не стоит всегда и всенепременно гнаться за краткостью. Первое, за чем стоит гнаться, это ясность и лаконичность, а краткость уже потом. Если вы в ущерб ясности гонитесь за краткостью, то так делать не стоит. В целом, рекомендую прочитать статью на сайте swift.org, которая называется API Design Guidelines, в которой Apple более подробно рассказывает про то, как лучше именовать сущности в языке Swift.
И, напоследок, давайте рассмотрим некоторые классы из стандартной библиотеки, которые помогут вам в выполнении домашнего задания. Полезные классы стандартной библиотеки, которые помогут вам при выполнении домашнего задания.

1.7.3. Полезные классы стандартной библиотеки

В завершении первой лекции давайте рассмотрим ряд полезных классов стандартной библиотеки, которые потребуются вам при выполнении домашнего задания к первой лекции.
Первое что мы рассмотри – это тип Date, или NSDate в objective-c, который позволяет вам очень гибко работать с датами. Для того, чтобы получить экземпляр объекта даты с текущим временем, достаточно просто его создать с пустым инициализатором:

```
let now = Date()
```

Вы получите объект, в котором будет текущие дата и время. Для того, чтобы сериализовать объект даты, например, в json, или в plist, или еще куда-нибудь, сам объект Date не годиться, для этого его нужно привести к какому-то более примитивному типу. Самый лучший кандидат для этого – это Unix timestamp. У класса Date есть способ получить unix timestamp из текущей даты:

```
let ts = now.timeIntervalSince1970
```

или создать объект даты из unit timestamp'a:

```
let now = Date(timeIntervalSince1970:1322252)
```

Часто возникает необходимость объект даты преобразовать в человекопонятную строку, в более превычные года, месяцы, дни, часы и так далее. Для этого сам объект Date уже не годиться. Для конверсии даты в строку и наоборот есть специальный класс DateFormatter. Для этого его нужно создать:

```
let fmt = DateFormatter()
```

задать ему формат:

```
fmt.dateFormat = "yyyy-MM-dd HH:mm:ss"
```

Дальше можно получить строку по дате:

```
let str = fmt.string(from: now)
```

или получить дату по строке:

```
let now2 = fmt.date(from: str)
```

Если вам требуется провести какую-то арифметику дат, например, к имеющейся дате добавить час, то для этого уже потребуется объект Calendar. В самом простом случае час к дате можно добавить так:

```
let nd = Calendar.current.date(byAdding: .hour, value: 1, date: now)
```

Тут стоит помнить, что календарей в iOS поддерживается несколько. Привычный нам грегорианский календарь лишь один из них. Есть еще, например, буддийский календарь, в котором сейчас шесть тысяч какой-то там год, или даже мусульманский, в котором тысяча четыресто какой-то там год. Тут важно помнить, что Calendar.current – это не всегда грегорианский календарь.
Идем дальше. Следующий полезный класс, который понадобится вам при выполнении домашнего задания – это UIColor. Он уже находится не в стандартной библиотеке, а в UIKit – библиотеке для разработки интерфейсов в iOS. Сам класс UIColor содержит ряд предопределенных значений. Например, если вам нужен красный цвет,то он там уже есть:

```
let red = UIColor.red
```

или зеленый:

```
let green = UIColor.green
```

Вы также можете захотеть получить свой собственный цвет. Тогда для этого есть свой специальный инициализатор через компоненты цвета:

```
let custom = UIColor(red: 34.0 / 255.0, green: 45.0 / 255.0, blue: 245.0 / 255.0,
alpha: 1.0)
```

Каждая из компонент цвета лежит в диапазоне от 0 до 1. Также вам может потребоваться получить компоненты цвета. Тогда для этого есть специальные функции:

```
var r: CGFloat = 0
var g: CGFloat = 0
var b: CGFloat = 0
var a: CGFloat = 0

color.getRed(&r, green: &g, blue: &b, alpha: &a)
```

При выполнении домашнего задания вам потребуется преобразовывать UIColor в строку, привычную, например, для web-разработчиков с решеткой, и наоборот.
Также вам потребуется преобразовывать объекты в json и обратно для работы с REST API в одной из следующих лекций. Вообще, с недавних пор в Swift появилась конструкция Codable, которая позволяет вам просто объявить вашу структуру как Codable, и с помощью класса JSONEncoder делать конверсию из специального класса Data (это просто абстракция набора байтов) сразу в нужный объект, и наоборот. Однако для целей курса в этом домашнем задании вам потребуется пройти несколько более сложным путем, который использовался до того, как появился Codable. Это необходимо для того, чтобы получить более глубокое понимание того, как работают extension'ы, и как можно работать со структурами данных.
До появления Codable разбор json'а представлял из себя двухшаговый процесс. Первый шаг – с помощью объекта JSONserialization из объекта класса Data мы получаем словарь или массив (зависит от корневого объекта в json):

```
do {
  let dict = try JSONSerialziation.jsonObject(with: data, options: [])
} catch {
}
```

Затем, итерируясь по этому словарю или массиву, мы строим наш объект. Эти шаги нужно будет проделать и вам при выполнении домашнего задания. Конверсия через JSONSerialization обратно в набор байт выглядит так:

```
let dict: [String: Any] = ...
do {
  let data = try JSONSerialization.data(withJSONObject: dict, options: [])
  let js = try JSONSerialization.jsonObject(with: data, options: [])
} catch {
}
```

Разобрать словарь и построить из него объект остается вам в качестве домашнего задания.
При выполнении домашнего задания вам потребуется путь в файловой системе с доступом на чтение и запись, как собственно, и способ эти файли читать и сохранять. Для этого в стандартной библиотеке есть специальный класс, который называется FileManager. В документации вы можете найти ряд чисто глобальных функций, которые также позволяют вам это делать, как и расширения к объектам класса Data. Но я все-таки рекомендую пользоваться классом FileManager, так как это более правильный способ это делать.
Для сохранения данных вам потребуется путь, в который можно писать, и из которого можно читать. Это может быть специальная директория Caches в sandbox'е вашего приложения. Путь до нее в виде объекта URL можно получить так:

```
let path =
FileManager.default.urls(for: .cachesDirectory, in: .userDomainMask).first
```

Директория Caches существует всегда, но вы можете захотеть создавать поддиректории. Для начала необходимо проверить ее существование:

```
var isDir: ObjCBool = false
if FileManager.default.fileExists(atPath: dirurl.path, isDirectory: &isDir),
isDir.boolValue {
}
```

Если она есть, то все хорошо, а вот если ее нет, то ее нужно создать. Для создания директорий и файлов у FileManager'а есть две функции – одна для создания директорий, другая для файлов. У функции, которая создает директорию:

```
try? FileManager.default.createDirectory(at: dirurl,
withIntermediateDirectories: true, attributes: nil)
```

есть специальный аргумент withIntermediateDirectories. Если он true, и в вашем пути есть несуществующие директории, то они будут созданы. Если же будет false, и несуществующие директории будут иметь место, то вы получите ошибку в виде исключения. Если файл и директория вам больше не нужны, их можно и нужно удалить:

```
try? FileManager.default.removeItem(at: dirurl)
```

На этом все. Я желаю вам удачи в выполнении домашних заданий и в дальнейших лекциях курса. Спасибо за внимание, и до свидания!