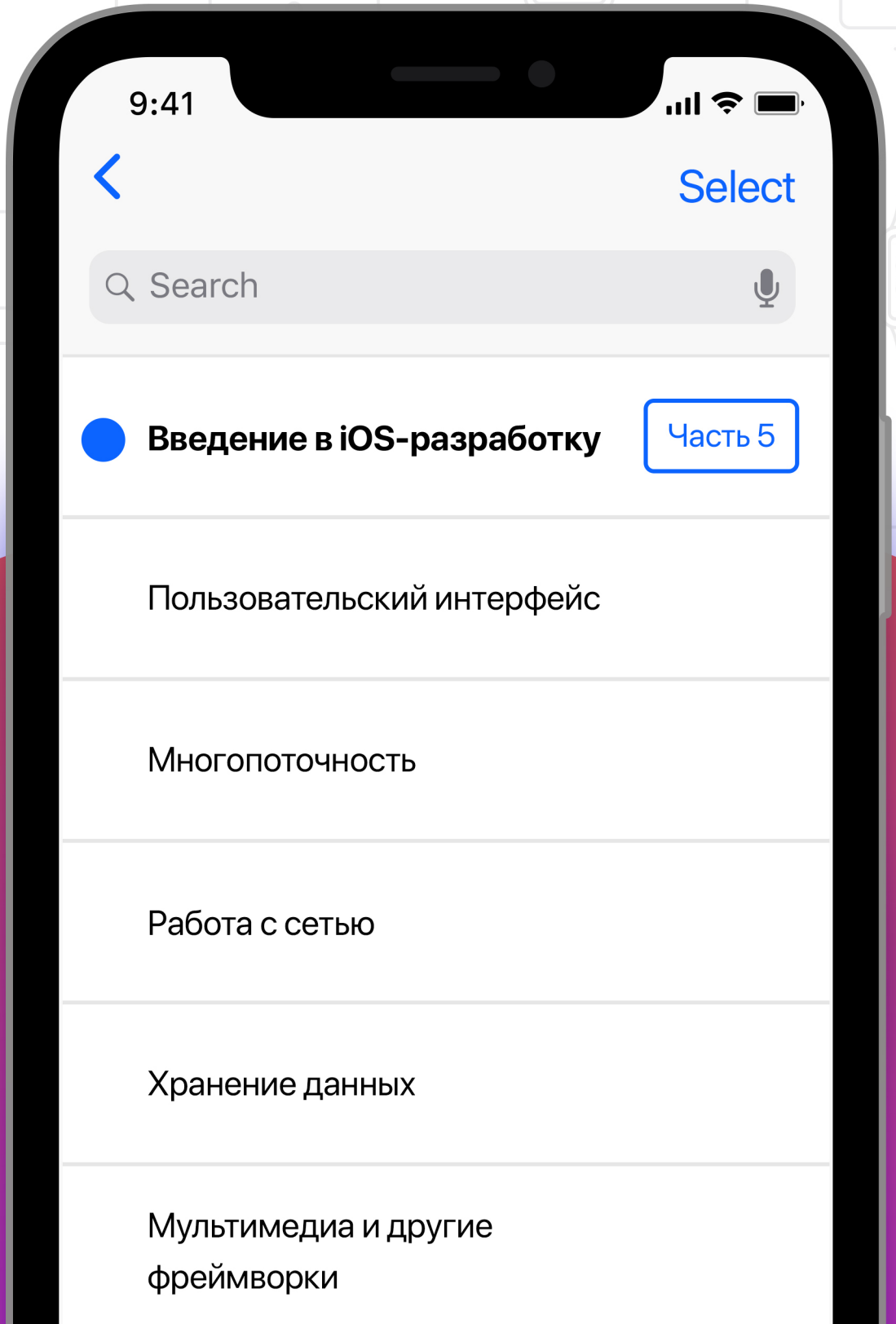
фонд развития
онлайн образования
eldf.ru

e·legion

academy.e-legion.com

# Программа iOS-разработчик

## Конспект



# Оглавление

# Глава 5

# НЕДЕЛЯ 5

## 5.1. Optionals

Приветствую! Эта лекция посвящена optional-значениям в Swift. Я расскажу о том, что это такое, как устроено внутри и для чего все это нужно.

### 5.1.1. Определение Optional

Optional — это контейнер для значений. Он может вернуть вам значение, содержащееся в нем, или сообщить, что значение отсутствует, и вернуть в этом случае nil. Например, при инициализации int-значения из строки у вас будет возвращен optional int, содержащий целочисленное значение в случае успеха, либо nil в случае неудачной инициализации. Пример на слайде.

```
1  let intFromString = Int("5")
2  print(intFromString) // Optional(5)
3  let failedIntFromString = Int("Five")
4  print(failedIntFromString) // nil
```

### 5.1.2. Объявление Optional

Для объявления переменных используются две формы записи: краткая и полная. Вариант с вопросительным знаком является предпочтительным, так как проще для записи и для восприятия.

```
1  let long: Optional<Int> = Int("5")
2  let short: Int? = Int("Five")
```

### 5.1.3. Устройство Optional

Что из себя представляет Optional? Это обычный enum, принимающий дженерик тип и содержащий два кейса: some и none. Соответственно и работать с ним можно, как с обычным enum.

```
enum Optional<T> {
     case some(T)
     case none
}
```

### 5.1.4. Nil значение

При объявлении переменной, если для нее не устанавливается значение, то оно автоматически определяется как nil. Для работы с Optional необходимо провести проверку на равенство nil, только после того как вы удостоверились, что переменная или константа содержат значение, вы можете работать с ней. Восклицательный знак используется для получения значения из Optional. Подобный прием называется Force unwrap. Попытка обращения к nil-значению при помощи Force unwrap приведет к аварийному завершению программы. Если при объявлении переменной или константы вы уверены в том, что значение точно будет установлено до первого обращения к ней, такую переменную можно пометить восклицательным знаком. Компилятор в этом случае позволит обращаться к переменной без дополнительных проверок и без Force unwrap. Однако такая переменная или константа все равно является optional-значением. В случае, если при работе с ней будет возвращено значение nil, вы также получите ошибку выполнения.

```
let counter: Int?         // nil
if counter != nil {
    counter!.increase()
}
```

### 5.1.5. Optional binding

Swift предлагает ряд инструментов для удобной и безопасной работы с optional-значениями. Optional Binding позволяет использовать if и while выражения для проверки значений в Optional. Перепишем ранее представленный пример: данное выражение не будет выполнено, если в counter содержится значение nil. Значение присвоенной переменной будет доступно только внутри выражения. При проверке можно присваивать значение нескольким переменным, а также выполнять

другие проверки, возвращающие bool. Если хотя бы в одном случае вернется false, код внутри выражения выполняться не будет.

```
if let nonOptionalCounter = counter {
    nonOptionalCounter.increase()
}
if let firstCounter = firstOptionalCounter, let
secondCounter = secondOptionalCounter,
firstCounter < secondCounter {
     print("Counters not nil. first counter <
     secondCounter")
}
```

### 5.1.6. Использование guard let

Для того чтобы с переменной можно было работать не только внутри выражения, можно использовать конструкцию guard let. Как упоминалось в предыдущих лекциях, из выражения guard необходимо выполнить ранний выход. Для этого, например, можно использовать return, continue, break или throw в зависимости от ситуации.

```
guard let nonOptionalCounter = counter {
    return
}
nonOptionalCounter.increase()
```

### 5.1.7. Nil-coalescing оператор

Еще одним способом, облегчающим работу с Optional, является использование nil-coalescing оператора, который обозначается с помощью двух вопросительных знаков. Это сокращенная форма для конструкции с использованием тернарного оператора. На слайде представлены два идентичных по значению выражения, но вторая запись проще для восприятия. В левой части оператора всегда должно находиться optional-значение, а в правой части значение того типа, которое содержится в optional-переменной.

```
nonOptionalCounter = counter != nil ? counter! : anotherCounter
nonOptionalCounter = counter ?? anotherCounter

```

```swift
let optional5: Int? = 5
let optionalNil: Int? = nil

var a = optional5 ?? 10 // 5
var b = optionalNil ?? 20 // 20
```

### 5.1.8. Optional Chaining

Теперь перейдем к optional chaining. Это процесс вызова метода, свойства или сабскрипта для optional-значения. Если значение Optional равняется nil, то и вызов функции, получение свойства и сабскрипта также вернет nil. Работа похожа на использование Force unwrap, только вместо восклицательного знака используется вопросительный. Результатом подобной операции всегда будет являться optional-значение. На слайдах показаны примеры использования Optional Chaining. Несмотря на то что свойства у объекта не являются optional-типами, результат все равно будет являться optional-значением, так как попытка обращения к свойству проходит по цепочке, в которой присутствует optional-значение. Chaining не ограничивается получением значения на одном уровне, вы можете продолжать получать по цепочке значение для свойств или вызывать методы. Этот прием работы можно использовать не только для получения, но и для установки значений. В этом случае часть кода справа от знака равенства не будет выполнена, если в левой части содержится nil. На примере показана попытка вызова функции, которая никогда не выполняется, так как у объекта в левой части отсутствует значение. Методы также можно вызывать с помощью Optional Chaining.

```swift
class SomeClass {
    struct InnerStruct {
        var variable: Int
    }
    var innerStruct: InnerStruct?
    }
let myClass = SomeClass()
var anotherOptional = myClass.innerStruct?.variable
myClass.innerStruct?.variable = 55

func getSomeValue() -> Int {return 5 }
s.innerStruct?.variable = getSomeValue()
```

### 5.1.9. Вызов методов

Любая функция в Swift возвращает либо значение либо void, даже если это не указано явно. В данном случае это будет Optional Void, Как и в предыдущих примерах мы можем воспользоваться условным оператором if для проверки выполнения функции.

```
class SomeClass {
    struct InnerStruct {
        func printSomething() {
            print("Что-то!")
        }
    }
var innerStruct: InnerStruct?
}
let myClass = SomeClass()
if myClass.innerStruct?.printSomething() != nil {
}
```

### 5.1.10. Subscript

При работе с сабскриптами необходимо обратить внимание на то, что вопросительный знак устанавливается до указания сабскрипта, а не после. Как и при работе со свойствами, можно не только получать, но и устанавливать значение для сабскрипта. Оба примера показаны на слайде. В случае получения optional- значения от сабскрипта, вопросительный знак ставим справа. Простым примером служит работа со словарями. При обращении к optional-значениям по сложной цепочке у вас не происходит дополнительного оборачивания Optional в Optional. То есть если вы должны были получить Optional String и для этого необходимо несколько раз использовать Chaining, то в конечном итоге вы все равно получите Optional String. На этом завершается краткий обзор Optional. В последующих уроках мы будем неоднократно использовать эти знания. А также демонстрировать другие приемы, облегчающие процесс написания программы с корректным и легко читаемым кодом.

```
class SomeClass {
    var arrayOfInts: [Int]?
}
let myClass = SomeClass()
myClass.arrayOfInts?[1] = 5
if let value = myClass.arrayOfInts?[1] {
    print("Some Value: \(value)")
```

```
}
```

## 5.2. Проверка типов

Приветствую. В этой лекции мы поговорим о том, как в Swift проверять и приводить типы. Приведение типов используется, если необходимо рассматривать объект как его сабкласс или родительский класс. Изменение типа на произвольный не имеет смысла, так как всегда будет заканчиваться неудачей.

### 5.2.1. Классы

Рассмотрим иерархию классов. Есть базовый класс для хранения имени устройства. От него наследуются классы, содержащие информацию о телефоне и наушниках. Они добавляют по одному полю. Также объявлена функция, выводящая информацию о девайсе. На вход она принимает родительский класс Device. В эту функцию можно передать устройство или любой его сабкласс.

```
class Device {
 var name: String
 init(name: String) {
    self.name = name
    }
}
class Phone: Device {
    var capacity: Int
    init(capacity: Int, name: String) {
       self.capacity = capacity
        super.init(name: name)
    }
}
```

```
class Headphones: Device {
    var resistance: Int
    init(resistance: Int, name: String) {
        self.resistance = resistance
        super.init(name: name)
    }
```

```
}
func printDescription(device: Device) {
    print("Device: \(device.name)")
}
```

### 5.2.2. Неявное приведение

Для этого выполняется неявное приведение типа. Оно выполняется только от более детализированного типа к менее детализированному. Например, от класса к его родительскому классу или от структуры к протоколу, который она поддерживает. Если бы функция принимала на вход телефон, то передать в нее базовые устройства было бы нельзя.

```
let phone = Phone(capacity: 256, name: "iPhone X")
let headphones = Headphones(resistance: 32,name: "Beats Studio3")

printDescription(device: phone)
printDescription(device: headphones)
```

### 5.2.3. Ошибка

Компилятор выдаст ошибку о невозможности конвертации типов.

```
func doSomethingWithPhone(phone: Phone) {
    print("\(phone.capacity)")
    // ...
}
let device = Device(name: "Some device")
doSomethingWithPhone(phone: device) // Error
```

### 5.2.4. Массив

Мы можем положить наши объекты в массив. При этом тип массива будет Device. Если мы возьмем объект из массива по индексу, то его тип тоже будет Device. Но мы можем проверить, является ли какой-то объект сабклассом определенного типа. Для этого используется оператор is, он возвращает результат проверки в виде bool. Мы можем воспользоваться этим оператором,

например, для того чтобы найти первый телефон в нашем массиве. Результат будет иметь тип Optional Device, так как функция first может вернуть nil, если элемент не найден. Оператор is не поможет нам обратиться к свойству Capacity полученного объекта, так как он просто осуществляет проверку.

```
let arrayOfDevices = [phone, headphones] // [Device]
arrayOfDevices[0] is Headphones // false
arrayOfDevices[1] is Device // true
let result = arrayOfDevices.first {$0 is Phone }
type(of: result) // Device?
```

### 5.2.5. As? As!

Для этого нам нужно явно привести объект к сабклассу. Такая операция называется Downcasting. В Swift для этого используются операторы as с вопросительным знаком и as с восклицательным знаком. As с вопросительным знаком возвращает опциональный тип. Если операция преобразования будет завершена с ошибкой, то результат будет nil. В примере на слайдах объект типа Phone приводится к Headphones. Так как он не является его сабклассом, то такое преобразование невозможно. As с восклицательным знаком, напротив, возвращает именно тот тип, котрый мы указали. Использовать его можно, только если вы уверены, что результат будет успешен, иначе ваше приложение завершится с ошибкой.

```
let optionalHeadprones = result as? Headphones
optionalHeadprones // nil
type(of: optionalHeadprones) // Headphones?
let nonOptionalPhone = result as! Phone
nonOptionalPhone.capacity // 256
type(of: nonOptionalPhone) // Phone
```

### 5.2.6. Использование If Let

Принудительного преобразования лучше избегать. Вместо него можно воспользоваться конструкцией if let для проверки значения. Рассмотрим пример. Допустим, нужно выводить разное описание для каждого сабкласса Device. Для этого необходимо привести тип и вывести значение полей. Теперь для каждого типа выводится свое описание. При этом, никаких ошибок произойти не может.

```swift
func printDetailedDescription(device: Device) {
    if let phone = device as? Phone {
        print("Phone: \(device.name) with capacity
        \(phone.capacity)")
 } else if let headphones = device as?
 Headphones {
        print("Headphones: \(device.name) with
        resistance \(headphones.resistance)")
    } else {
        print("Device: \(device.name)")
    }
}
printDetailedDescription(device: phone) // Phone: iPhone X with capacity 256

printDetailedDescription(device: headphones) // Headphones: Beats Studio3 with
                                                        //resistance 32
printDetailedDescription(device: device) // Device: Some device
```

### 5.2.7. Upcasting

Также в Swift есть простой оператор as. Его использование ограничено ситуациями, когда уже на этапе компиляции известно, что результат будет успешным. Например, при преобразовании типа вверх по иерархии классов или при взаимодействии с некоторыми типами Objective-C. Рассмотрим оба случая. Первый называется Upcasting. В большинстве случаев Swift сам выполняет такое преобразование, однако оператор as может оказаться полезным в случае неоднозначностей. На слайдах вы видите две функции с одинаковым названием, но разными типами. Swift сам вызовет нужную, исходя из типа аргумента. Так как передаваемый объект имеет тип phone, то вызовется соответствующая функция. Однако в некоторых случаях нам нужно изменить это поведение.

```swift
func process(_ device: Device) {
    print("process as Device")
}
func process(_ phone: Phone) {
    print("process as Phone")
}
process(phone) // process as Phone
process(phone as Device) // process as Device
```

### 5.2.8. Bridging

Для этого можно воспользоваться оператором as, так как результат преобразования не может быть nil. Второй пример использования этого оператора — это бриджинг между типами Swift и Objective-C. Бриджинг — это автоматическое преобразование типов. Вы можете ознакомиться со списком типов, поддерживающих эту возможность, по ссылке в материалах к лекции. Мы рассмотрим работу со строками. Для того чтобы присвоить строку к NSString нужно просто указать этот тип после оператора as. То же самое действует и в обратную сторону. Преобразование типов в некоторых случаях необходимо. Разрабатывая приложение, вы обязательно с ним столкнетесь, однако если вам приходится использовать, задумайтесь, возможно, есть другой способ выполнить задачу и не завязываться на тип данных.

```
let swiftString = "This is string"
let objcString: NSString = swiftString as NSString
let sameSwiftString: String = objcString as String
```

### 5.2.9. Использование протокола

Давайте рассмотрим, как можно переделать функцию, выводящую разное описание для каждого типа, чтобы все было в стилистике Swift. Объявим протокол. Его будут реализовывать все объекты, информацию о которых нужно выводить. Так как через расширение нельзя переопределять методы в сабклассах, придется переделать первоначальное объявление классов.

```
protocol Printable {
    func detailedDescription() -> String
}
class Device: Printable {
    var name: String
    init(name: String) {
        self.name = name
    }
    func detailedDescription() -> String {
            return "Device: \(device.name)"
    }
}
```

### 5.2.10. Использование протокола

Функция *printDetailedDescription* будет принимать объект типа printable и просто выводить результат выполнения метода, описанного в протоколе. Теперь можно передавать функцию, любой printable тип. При этом мы избежали не очень красивой проверки типа.

Функция *printDetailedDescription* стала зависеть только от абстрактного протокола и не знает деталей реализации других типов. Таким образом, обеспечивается слабое сцепление классов.

```
class Phone: Device {
     var capacity: Int
     init(capacity: Int, name: String) {
         self.capacity = capacity
         super.init(name: name)
     }
     override func
     detailedDescription() -> String {
         return "Phone: \(device.name)
         with capacity
         \(phone.capacity)"
     }
}
```

```
class Headphones: Device {
     var resistance: Int
     init(resistance: Int, name: String) {
        self.resistance = resistance

         super.init(name: name)
     }
     override func
     detailedDescription() -> String {
         return "Headphones:
         \(device.name) with
         resistance
         \(headphones.resistance)"
     }
}
```

### 5.2.11. Универсальная функция

Благодаря тому, что использовались протоколы, а не обычные наследования, мы легко можем передать в эту функцию любой тип, в том числе не сабкласс Device.

```
func printDetailedDescription(_ printable: Printable) {
    print(printable.detailedDescription())
}

printDetailedDescription(phone) // Phone: Some device with capacity 256

printDetailedDescription(headphones) // Headphones: Some device with resistance 32

printDetailedDescription(device) // Device: Some device
```

### 5.2.12. Расширение UIView

В примере поддержка Printable добавлена в системный класс UIView, в котором для формирования описания используется стандартный метод Description. На этом лекция по проверке типов подходит к завершению. Мы рассмотрели, как можно проверить и приводить типы в Swift. Также мы затронули тему взаимодействия Swift и Objective-C, но более подробно об этом мы расскажем на следующих курсах.

```
extension UIView: Printable {
    func detailedDescription() -> String {
        return self.description
    }
}
let view = UIView(frame: CGRect(x: 10, y: 20, width: 4, height: 5))
printDetailedDescription(view) // <UIView: 0x7fac1950b390; frame = (10 20; 45);
                                                    layer = <CALayer: 0x60800002f380>>
```

## 5.3. Exceptions

Данная лекция посвящена работе с исключениями в Swift. Как и во многих других языках, в Swift есть возможность использовать throw для того, чтобы выбрасывать исключения, или exception. Они помогают передать вызывающей стороне причину, по которой выполнение ко-

да было прервано. При этом в Swift у исключений есть свои особенности, помогающие писать безопасный код.

### 5.3.1. Error

Ошибка может быть представлена любым типом, поддерживающим protocol error. Он не предъявляет никаких требований. В качестве типа отлично подходит перечисление. На слайде изображен пример такого перечисления. Каждый кейс представляет одну из ошибок. Благодаря associated values, мы можем приложить к ошибке какие-то данные, например, код ошибки.

```
enum ProcessingError: Error {
      case incorectInput
      case noConnection
      case internalError(errorCode: Int)
}
```

### 5.3.2. Выбрасывание исключения

Для того чтобы бросить исключение, используется ключевое слово throw. После него указывается экземпляр любой сущности, поддерживающий protocol error.

```
throw ProcessingError.internalError(errorCode: 2)
```

### 5.3.3. Бросание исключения из функции

Для того чтобы в функции можно было использовать исключения, ее нужно пометить ключевым словом throws. Таким образом, и без чтения документации к ней будет понятно, что она может бросить исключение. Компилятор будет контролировать, перехватили ли вы исключение при вызове и не пытаетесь ли вы бросить его из функции, не помеченной throws.

```
func canThrowErrors() throws -> String {
    throw ProcessingError.internalError(errorCode: 2)
}
```

### 5.3.4. Исключение в инициализаторе

Помимо обычных функций и методов, исключения можно использовать и в инициализаторах.

Принцип такой же. Простой пример с проверкой на нулевой входной параметр приведен на слайде.

```
struct MyStruct {
    init(parameter: Int) throws {
        guard parameter != 0 else {
            throw ProcessingError.incorectInput
        }
    }
}
```

### 5.3.5. Передача выше по стеку

Теперь поговорим о том, как перехватывать исключения. Есть четыре варианта их обработки. Первый — никак с ними не взаимодействовать и позволить им подняться выше по стеку вызовов. Для этого придется пометить throws и вызывающую функцию, так как она теперь тоже может передавать исключения во внешнюю область видимости. Кроме этого, при вызове функции, помеченной throws, нужно добавлять ключевое слово try.

```
func canThrowErrorsTo() throws {
    try canThrowErrors()
}
```

### 5.3.6. Catch

Второй способ — классический catch. Функция вызывается внутри блока do с указанием ключевого слова try. Далее указываются шаблоны для определения типа исключения. Если шаблон отсутствует, то считается, что этот catch ловит любые исключения. Такой кейс на слайде приведен последним. Подробная информация о шаблонах приведена в лекции Pattern matching. Благодаря им мы можем очень гибко настроить перехват нужных исключений. Если выявлено несколько совпадений, то сработает первое из них. Если ни один из шаблонов не подошел, то исключение будет передано во внешнюю область видимости. К сожалению, на данный момент нет никакой возможности указать, какой тип исключений может бросить функция. Поэтому, если вы хотите перехватить все исключения и не передавать их выше, то вам придется указать в конце catch без шаблонов или шаблон, который совпадает с любой ошибкой. В противном случае Swift будет считать, что вы перехватили не все возможные исключения и будет требовать от вас пометить вызывающую функцию как throws.

```swift
do {
    try canThrowErrors()
} catch ProcessingError.incorectInput {
    // ...
} catch ProcessingError.noConnection {
    // ...
} catch ProcessingError.internalError(let errorCode)
where errorCode == 1 {
    // ...
} catch ProcessingError.internalError(let errorCode) {
    // ...
} catch let anotherError {
    // ...
}
```

### 5.3.7. Try?

Следующий вариант обработки исключений — конвертация возвращаемого значения в опциональный тип. Для этого используется ключевое слово try с вопросительным знаком. В этом случае, если функция бросает исключения, то результат будет равен nil, то есть результат выполнения этой функции становится опциональным. Если функция ничего не возвращает, то брошенное исключение просто игнорируется. В примере мы вызываем функцию, возвращающую строку. Так как она бросила исключение, мы получим в результате nil.

```swift
let result = try? canThrowErrors()
type(of: result) // Optional<String>
```

### 5.3.8. Try!

И последний вариант — это принудительное игнорирование исключения с помощью ключевого слова try с восклицательным знаком. Использовать его можно только в том случае, если вы уверены, что исключение действительно не возникнет, иначе ваша программа завершит выполнение с ошибкой.

```swift
try! canThrowErrors()
```

### 5.3.9. Использование guard Let и If Let

Так же, как и force unwrap, этого способа лучше избегать. Воспользуйтесь лучше try с вопросительным знаком и конструкцией guard let или if let.

```
guard let result = try? canThrowErrors() else {
    return
}

// или

if let result = try? canThrowErrors() {
    // ...
} else {
    // ...
}
```

### 5.3.10. Исключение - это Enum

Пару слов о производительности. В отличие от других языков программирования, в Switf не используются сложные алгоритмы раскручивания стека вызова для передачи исключений. Для компилятора брошенное исключение — это то же самое, что и обычный return. Реализуется это с помощью перечисления. Оно хранит два состояния. В случае ошибки в associated value может храниться любой объект, поддерживающий protocol error, в случае успеха — значение, возвращаемое из функции.

```
enum Result<T> {
    case fail(Error)
    case success(T)
}
```

### 5.3.11. Использование своего типа

Можно переписать функцию canThrowError с помощью этого перечисления. Разумеется, если мы реализуем собственное перечисление для ошибок, то мы лишаемся удобных конструкций, таких как try и do catch, а также проверок компилятора. Благодаря простой реализации исключений в Swift не приведут к проблемам с производительностью. Учитывая все это, нет смысла

придумывать свой способ обработки ошибок. Например, передавать их через INOUT parameter, как это было реализовано в Objective-C.

```
func canThrowErrorUsingEnum() -> Result<String> {
    return .fail(ProcessingError.internalError (errorCode: 2))
}
```

### 5.3.12. Метод для создания ссылки в Objective-C

Если у вас есть опыт программирования на Objective-C, то вы знаете, что исключения в нем перехватываются редко. Для передачи ошибок используется класс NSError. На слайде в качестве примера приведен метод для создания ссылки на файл. NSError передается как указатель на указатель. Если что-то пойдет не так, то в переменной, которую мы передали в этот метод, будет находиться описание ошибки. В дополнение к этому метод возвращает BOOL — значение, указывающее, было ли создание ссылки успешным. Для определения успешности выполнения используется именно оно, так как error может содержать некоторый мусор, даже если ошибок не возникло. В Swift такой подход смотрелся бы чужеродно. Поэтому при генерации интерфейсов для Swift из Objective-C применяются некоторые преобразования.

```
- (BOOL)createSymbolicLinkAtURL:(NSURL *)url
withDestinationURL:(NSURL *)destURL
error:(NSError * _Nullable
*)error;
```

### 5.3.13. Метод для создания ссылки в Swift

Во-первых, NSError заменяется на исключение. Оно будет содержать свойство localized description. В нем находится описание ошибки, полученное из NSError. Во-вторых, возвращаемый BOOL заменяется на VOID. Если метод возвращает какой-то другой тип, то он останется.

```
    open func createSymbolicLink(at url: URL,
    withDestinationURL destURL: URL)
throws
```

### 5.3.14. Обработка исключения

На слайде представлен вызов метода, создающего ссылку на файл. Он обернут в конструкции do catch, ведь в Swift эта функция бросает исключения. Так как в аргументах передан некорректный путь, то мы увидим соответствующую ошибку.

```
do {
    try FileManager.default.createSymbolicLink(at:
    URL(string: "/")!,
withDestinationURL: URL(string: "/")!)
} catch let error {
    print("\(error.localizedDescription)")
    // The file couldn't be saved because the specified URL type isn't supported.
}
```

### 5.3.15. Exception в замыкании

Представьте ситуацию, в которой вам необходимо применить какую-то логику к целой коллекции, например, создать несколько ссылок на файл. Сделать это можно с помощью массива и forEach. Однако обратите внимание, что замыкание, которое мы передали в forEach, может бросить исключение. Скорее всего, мы не захотим ловить и обрабатывать его прямо внутри замыкания, как это показано на слайде. Для того чтобы передать ошибку выше, придется помечать методы вроде forEach, map и reduce и так далее ключевыми словами throws. Из-за этого придется при их вызове всегда использовать try, даже если замыкание не может бросить исключение. Второй вариант — это создать две версии forEach. Одну — для обычных замыканий, а вторую — для тех, которые используют исключения. Очевидно, что это тоже не очень хорошая идея из-за дублирования кода.

```
let linkDestinations = [URL(string: "/somePath/file")!,
        URL(string: "/somePath2/file")!,
        URL(string: "/anotherPath/file")!]
linkDestinations.forEach {
 url in

 do {
     try FileManager.default.createSymbolicLink(at:
     URL(string: "/")!, withDestinationURL: url)
 } catch let error {
    // ...
```

```
12  }
13  }
```

### 5.3.16. Объявление forEach

Для решения этой проблемы было добавлено ключевое слово rethrows. На слайде вы видите объявление метода forEach. Такая конструкция означает, что метод принимает замыкание, которое может бросать исключение. Однако сам метод будет являться throws, только если замыкание действительно их использует. В этом случае нам, как обычно, нужно воспользоваться try.

```
1  func forEach(_ body: (Element) throws -> Void)
2  rethrows
```

### 5.3.17. Использование Rethrows

На слайде представлен пример. Замыкание может бросить исключение, поэтому вызов forEach обернут в конструкцию do catch.

```
1  do {
2      try linkDestinations.forEach {
3          url in
4
5          try FileManager.default.createSymbolicLink(at: URL(string: "/")!,
6  withDestinationURL: url)
7      }
8  } catch let error {
9      // ...
10 }
```

### 5.3.18. В данном случае исключений нет

Если же замыкание не может бросить исключение, то компилятор разрешит нам не обрабатывать ошибки, так как они не могут возникнуть.

```
linkDestinations.forEach {
    url in

    print("\(url)")
}
```

### 5.3.19. Defer

Напоследок рассмотрим ключевое слово defer. Оно позволяет вам выполнить код при выходе из текущей области видимости. Он будет выполнен, даже если было брошено исключение. Такую конструкцию удобно использовать, например, для освобождения каких-то ресурсов. При этом она не привязана к do catch или try. Вы можете использовать defer для любых ситуаций. Он будет вызван при выходе из области видимости.

```
defer {
    // Какие-то операции
}

try? canThrowErrors()
```

### 5.3.20. Простейший итератор

Помните, мы использовали его создания простейшего итератора? При вызове метода next сначала текущее значение base сохраняется в качестве результата выполнения функции, а затем выполняется defer. Таким образом, мы избавились от ненужной дополнительной переменной для сохранения текущего значения base. На этом все. Не забывайте, что исключения могут значительно изменить логику выполнения вашего приложения. Нужно внимательно следить за тем, что все ваши объекты находятся в правильном состоянии, даже если возникло исключение.

```
struct MultiplicityIterator: IteratorProtocol {
    var base = 1

    public mutating func next() -> Int? {
        defer {base += base }
        return base
    }
}
```

## 5.4. Pattern Matching

Привет. В прошлых лекциях мы уже упоминали паттерны, например, когда рассказывали про оператор Switch. В этой лекции мы рассмотрим эту тему подробнее. Паттерн представляет собой структуру простого или составного типа. Они делятся на два типа. Первые совпадают с любыми значениями, но вы можете дополнительно указать их тип. А вторые используются для нахождения конкретных значений. Давайте рассмотрим подробнее.

### 5.4.1. Wildcard pattern

К первому типу относится Wildcard pattern. Он совпадает с любыми значениями любых типов. При этом у вас уже не будет возможности получить эти данные. Такой шаблон нужен, когда вы хотите проигнорировать какое-то значение, например, если результат выполнения функции вам не нужен. Просто так вызвать такую функцию нельзя. Swift будет выдавать предупреждение о неиспользованном результате. Также можно использовать его в for... in цикле, если вам нужно просто выполнить какой-то код несколько раз, а порядковые номера не важны.

```
func someCalculation() -> Int {return 5 }
_ = someCalculation()

for _ in 1...10 {
    // ...
}
```

### 5.4.2. Identifier pattern

Identifier pattern совпадает с любым значением и связывает его с константой или переменной.

```
let someString = "string"
```

### 5.4.3. Value binding pattern

Value binding pattern состоит из ключевого слова let или var и шаблона после него. В результате все идентификаторы в паттерне будут связаны с константой или переменной. Благодаря ему

можно упростить запись. Например, код на слайдах эквивалентен, но второй вариант более лаконичен.

```
let tuple = (1, 2)
if case (let a, let b) = tuple {
    print("a: \(a), b: \(b)")
}
if case let (a, b) = tuple {
    print("a: \(a), b: \(b)")
}
```

### 5.4.4. Tupel pattern

Следующий вид — это Tupel pattern. Мы его использовали в предыдущем примере, но мы не уточняли тип данных. Это можно сделать, указав его после двоеточия.

```
let (a, b): (Int, Int) = (1, 2)
```

### 5.4.5. Enumeration case pattern

Далее мы рассмотрим второй тип паттернов. Начнем с Enumeration case pattern. Он используется в Swift и в других case-конструкциях. На слайдах вы видите простой пример использования Switch для определения значения перечисления. При этом каждый паттерн в отличие от предыдущих примеров ожидает конкретное значение, а не просто структуру или тип данных.

```
enum SomeType {
     case type1
     case type2
     case type3
}
let type = SomeType.type1
```

```
switch type {
case .type1:
    // ...
    break
```

```
5
6   case .type2:
7       // ...
8       break
9
10  case .type3:
11      // ...
12      break
13  }
```

### 5.4.6. Optional pattern

Для получения значений из опционального типа можно использовать Optional pattern. По сути это тот же Enumeration case pattern, только с синтаксическим сахаром. Состоит он из Identifier pattern и вопросительного знака. Приведенные на слайде конструкции эквивалентны.

```
1   let optValue: Int? = 654
2   if case let value? = optValue {
3       // ...
4   }
5   if case let .some(value) = optValue {
6       // ...
7   }
```

### 5.4.7. Type casting pattern

Type casting pattern используется в Switch. Есть два таких паттерна: is и as. Их поведение похоже на операторы is и as. Они совпадают, только если тип такой же или унаследован от указанного. На слайдах — простой пример, как это можно использовать на практике.

```
1   let someValue: Any = 0.0
2
3   switch someValue {
4   case 0 as Double:
5       // ...
6       break
7   case 0 as Int:
```

```
        // ...
        break
case let anotherInt as Int:
        // ...
        break
default:
        // ...
        break
}
```

### 5.4.8. Expression pattern

Ну и последний вид — это Expression pattern. Он также используется в Switch операторе. Для проверки совпадения используется оператор ~=. Помимо простого сравнения значений можно проверить целый диапазон или переопределить оператор и предоставить свою логику. В примере на слайдах объявлен оператор для целого числа и строки. В нем строка преобразуется к числу и сравнивается с шаблоном. Теперь можно в Switch передавать строку с числом, а в качестве паттернов использовать целые числа. Паттерны — это очень мощная возможность языка. Вы будете часто их использовать, особенно при работе с кортежами и в операторе Switch. Надеюсь, после нашей лекции вы стали лучше понимать, как Swift обрабатывает такие конструкции.

```
func ~= (pattern: Int, value: String)
    -> Bool {
     if let intValue = Int(value) {
        return intValue == pattern
     }
     return false
}
```

```
switch "123" {
case 321:
    // ...
    break

case 123:
     // ...
     break
```

```
9
10  default:
11      // ...
12      break
13  }
```

## 5.5. Определение generics

### 5.5.1. Перегруженный метод

Сегодня мы рассмотрим, как избежать дублирования кода для общей логики, как отказаться от перегрузки методов, функции и создавать типы данных, не зависящие от типизации. Использование дженериков позволяет нам создавать функции и типы, которые будут работать с любым типом данных в зависимости от наших требований. Благодаря этому, мы можем выделить абстрактную логику и избежать дублирование кода, написанного для использования с разными типами данных. Дженерики не являются новшеством в программировании. Мы можем встретить их в других языках, таких как C++, Java, *C#*, D. Этот подход к программированию называется обобщённое программирование. Представим, что нам необходимо реализовать функцию, описывающую переданный аргумент в определённом формате. Без использования дженериков нам пришлось бы перегружать функцию для каждого типа данных. Эти функции выполняют один и тот же код, не зависящий от типа данных, поэтому, используя дженерики, мы можем написать единую реализацию для метода. Функция используется PlaceholderType. Что это такое? Это псевдотип данных, так называемый Placeholder. Мы сообщаем функции или методу, что она работает с чем-то, что придёт извне. Так как логика не затрагивает методов и свойств определённого типа, то нам и не важно в реализации функции, с каким типом данных мы работаем. Вместо PlaceholderType может быть любой текст — t, anytype — однако, лучше использовать информативные имена. Название типа описывается в UpperCamelCase формате.

```
1  func customDescription(_ arg: Int) -> String {
2      return "Аргумент: \(arg)"
3  }
4  func customDescription(_ arg: (Double, Bool)) ->
5  String {
6      return "Аргумент: \(arg)"
7  }
8  func customDescription<PlaceholderType>(_
9  arg: PlaceholderType) -> String {
```

```
10      return "There is our arg: \(arg)"
11  }
```

### 5.5.2. Определение

Объявления всех плейсхолдеров происходят сразу после имени функции в треугольных скобках и через запятую. Можно объявлять такое количество плейсхолдеров, которое вам будет нужно.

```
1   //func имяФункции<Плейсхолдер1, Плейсхолдер2>
2   //(параметр1: Плейсхолдер1, параметр2: Плейсхолдер2) -> Плейсхолдер2
```

### 5.5.3. Реализация стека

Кроме функций с дженерик-параметрами Swift позволяет определять свои дженерик-типы данных. Это могут быть классы, структуры или перечисления, которые будут работать с любым типом данных аналогичным образом как массивы или словари. На слайде показана структура, реализующая стек. Реализация основывается на использовании массива, в котором хранятся значения. В нашем примере мы жёстко типизировали контейнер для хранения Int. Такая реализация позволяет работать нам только с одним типом данных. Для работы со строками нам потребуется написать аналогичную структуру с типизацией String или же использовать общий тип Any.

```
1   struct StackOfInteger {
2       var items = [Int]()
3       mutating func push(_ item: Int) {
4           items.append(item)
5       }
6       mutating func pop() -> Int {
7           return items.removeLast()
8       }
9   }
```

### 5.5.4. Использование Any

Использование Any — не лучшее решение в данном случае. Каждый раз при работе с данными

нам придётся использовать приведение типа.

```
struct StackOfAny {
     var items = [Any]()
     mutating func push(_ item: Any) {
        items.append(item)
     }
     mutating func pop() -> Any {
        return items.removeLast()
     }
}
let item = "Contained info"

var stackOfAny = StackOfAny(items: [item])
let popedAnyItem = stackOfAny.pop() // Any тип
let stringValue: String = popedAnyItem as! String // String тип
```

### 5.5.5. Дженерик параметры

Лучшим решением будет создание структуры с дженерик-параметрами. Так как массивы Swift могут содержать любые типы данных одного типа, то нам не важно, что будет на входе. Мы сообщаем структуре об используемом типе данных с помощью ContainedType.

```
struct StackOfGenerics<ContainedType> {
     var items = [ContainedType]()
     mutating func push(_ item: ContainedType) {
        items.append(item)
     }
     mutating func pop() -> ContainedType {
        return items.removeLast()
     }
}
```

### 5.5.6. Дженерик параметры

Теперь при обращении к стеку мы получаем типизированное значение, которое можно использовать без приведения типа. Есть и отрицательные моменты при использовании дженери-

ков. Например, снижение производительности во время исполнения. Это связано с дополнительной обработкой кода, который будет сгенерирован. Что не так важно при редком обращении к дженерик-коду, но ощутимо при работе с большими модулями, где дженерик-код используется повсеместно. Например, стандартные библиотеки.

```
var stackOfGenerics = StackOfGenerics(items: [item])
let popedGenericItem = stackOfGenerics.pop() // String тип
let result = (popedAnyItem as! String) + popedGenericItem
```

### 5.5.7. @_specialize

Для решения этой задачи был введён атрибут @_specialize, который подсказывает оптимизатору, какие типы данных будут использоваться чаще всего. Атрибут применяется только для функции. Такое решение требует балансировки, времени компиляции, размера бинарного файла и производительности в режиме исполнения. Не стоит использовать данный атрибут повсеместно. Также важно помнить, что поддержка этого атрибута не гарантируется в следующих версиях языка. Следующая лекция будет посвящена дженерик-констрейтам, с помощью которых мы можем ограничивать дженерик-параметры различными условиями.

```
struct StackOfGenerics<ContainedType> {
     var items = [ContainedType]()

     @_specialize(where ContainedType == Int)
     mutating func push(_ item: ContainedType) {
        items.append(item)
     }

     @_specialize(where ContainedType == Int)
     mutating func pop() -> ContainedType {
        return items.removeLast()
     }
}
```

## 5.6. Generic constraints

Всем привет! В этом видео мы продолжаем рассказывать о дженериках. В прошлой лекции мы создали структуру, в которой реализовали стек-хранилище.

### 5.6.1. Использование constraints

Но что, если нам необходимо добавить метод, сравнивающий переданный аргумент с первым элементом стека? Для этого нам необходимо быть уверенным, что тип данных значений в нашем стеке реализует протокол Equatable. То есть значения могут сравниваться между собой. Дженерик-код имеет ограничители, называемые constraints. Это дополнительные условия вхождения в нашу функцию, класс, перечисление. Мы меняем ранее созданную структуру. Мы указываем, что ContainedType должен реализовывать протокол Equatable. Такое же ограничение можно добавить и для метода, выполняющего сравнение элементов. Создаём расширение для структуры и описываем метод. В этом коде Equatable является указанием, каким типом данных ограничивается наша функция. Это может быть либо имя класса, либо протокола, при этом учитывается иерархия наследования.

```
struct StackOfGenerics<ContainedType:
Equatable> {... }
extension StackOfGenerics where
ContainedType: Equatable {
    func isItemOnTop(_ item: ContainedType)
    -> Bool {
        return items.last == item
    }
}
```

### 5.6.2. Дополнительные условия

Часто возникают ситуации, в которых необходимо ввести дополнительные условия. К примеру, нам необходима функция, которая сравнит переданную коллекцию с нашим стеком и вернёт значение true, если они эквивалентны. Для реализации нам необходимо, чтобы элементы стека и элементы переданной коллекции реализовывали Equatable протоколы, что позволит нам сравнивать их. Так же элементы двух коллекций должны быть одного типа. Для этого мы создали дополнительный ограничитель, использующий ключевое слово where.

```
extension StackOfGenerics where ContainedType:
Equatable {
 func isCollectionEqualToStack<CollectionType:
 Collection>(_ newColection: CollectionType) -> Bool
    where CollectionType.Iterator.Element ==
    ContainedType {
```

```
7           guard items.count ==
8           newColection.count as! Int else {return false }
9           for (index, item) in newColection.enumerated() {
10              if items[index] != item {
11                  return false
12              }
13          }
14          return true
15      }
16  }
```

### 5.6.3. Использование в функции

Теперь рассмотрим перегрузку функций с дженерик-параметрами. Мы создаём две функции. Какая из функции будет вызвана при передаче UILabel в качестве параметра? Важно помнить, что Swift имеет множество правил для определения порядка вызова функций. Так как перегрузка функций разрешается статично во время компиляции, основываясь на типах значений, а не на самих значениях, то во время выполнения будет вызвана функция, получающая в качестве параметра тип UILabel. В языке Swift мы часто используем протоколы, которые также можно описывать используя дженерик-подход. Но есть отличие. Мы не можем определить протокол с дженерик-параметрами, как мы это делаем для функций, структур, классов. Для работы протоколов с дженерик-параметрами существует associated types. О том, как с ними работать, я расскажу в следующей лекции.

```
1  func log<PlaceHolderType: UIView>(_ view:
2                       PlaceHolderType) {... }
3  func log(_ view: UILabel) {... }
```

## 5.7. Associated Types

Всем привет. В этом видео мы продолжим тему дженериков и поговорим про ограничители для протоколов. Так как протоколы не являются полноценными типами данных, то мы не можем использовать дженерик-типы, как это делали для функций и классов. Протоколы описывают интерфейс без реализации логики, поэтому нам неважно, как будет построена логика вычисления в объектах, реализующих протокол. Нам важно объявить плейсхолдеры и далее использовать

их в написании сигнатур. А задачу типизации будет решать тот объект, который реализует наш протокол.

### 5.7.1. Объявление

Для использования дженерик-кода в протоколах мы создаем associatedtype. плейсхолдер, с которым мы ассоциируем наш протокол. Объявляется он с помощью ключевого слова associatedtype. При реализации протокола обычно нет необходимости указывать, какой тип данных мы будем использовать. Эта информация будет определена Swift самостоятельно.

```
protocol Printable {
    associatedtype PrintableType
    associatedtype OtherPrintableType
    func printValues(_ arg1: PrintableType,
    and arg2: OtherPrintableType)
}
```

### 5.7.2. Пример реализации протокола

Пример на слайде описывает класс и реализацию метода для поддержки протокола. Типы данных аргументов — Bool и Int — мы определили в классе. И именно эта информация будет использоваться Swift при реализации протокола. Так как в классе явно определены типы данных, то мы опустили типизирование нашего протокола.

```
class Document: Printable {
    func printValues(_ arg1: Bool, and arg2: Int) {
    }
}
```

### 5.7.3. Typealias

Но что, если хотим явно определить тип данных, с которым будет работать наш объект, реализующий протокол? В этом случае мы используем typealias. Осталось рассмотреть пример использования constraint'ов для associatedtype.

```
protocol ViewModel {
    associatedtype ContextType
```

```
    var context: ContextType?
    init(with context: ContextType?)
}
```

```
class ScheduleViewModel:
ViewModel {
    typealias ContextType = Int
    var context: ContextType?
    required init(with context:
    ContextType?) {
        self.context = context
    }
}
```

### 5.7.4. Ограничения для типов

В данном случае используются два плейсхолдера — Item и Iterator.

Аналогично дженерик-параметрам в контейнерах мы ограничили тип Iterator дополнительными проверками. Теперь использовать наш протокол можно с любыми типами данных, где Iterator будет поддерживать IteratorProtocol, а каждый элемент Iterator принадлежит типу Item. Разобраться с теорией будет проще на практических примерах, чем мы и займемся в следующих лекциях.

```
protocol Container {
    associatedtype Item
    mutating func append(_ item: Item)
    var count: Int {get }
    subscript(i: Int) -> Item {get }
    associatedtype Iterator: IteratorProtocol
    where Iterator.Element == Item
    func makeIterator() -> Iterator
}
```

## 5.8. Использование generics

Приветствую на лекции, посвящённой устройству дженериков и associated type в протоколах.

### 5.8.1. Устройство Generic

Рассмотрим пример функции, приведённой на одном из видео с WWDC. В функции min есть один плейсхолдер T, на него наложено только одно ограничение: он должен поддерживать сравнение. Как компилятор будет обрабатывать эту функцию? Ему неизвестен размер переменной, которая будет передаваться в функцию, ему неизвестно расположение реализации метода сравнения, ведь для каждого типа оно будем своим. Эта проблема, как и многие другие в программировании, решается с помощью ещё одного уровня абстракции. Когда компилятор доходит до такой функции, он оборачивает дженерик-значения в контейнер. Они имеют фиксированный размер, а если значение в него не влезет, то контейнер будет хранить указатель на него, а само значение будет лежать в куче. Так решается первая проблема. Дополнительно к этому компилятор создаёт несколько witness tables: одну value witness table и по одной protocol witnes table для каждого протокола, ограничивающего плейсхолдер. Witness table содержит указатели на фундаментальные операции над конкретным типом, например, создание, копирование или уничтожение. Также в этой таблице содержится реальный размер типов. В нашем примере дженерик-параметр T будет иметь одну protocol witness table. В ней будут содержаться указатели на реализацию всех методов сравнения в переданном типе. Каждое обращение к такому методу во время выполнения будет проходить через таблицу. В результате получится примерно такой код. В функцию передаются оба параметра, обёрнутые в контейнер, и две таблицы. Как вы видите, протоколы тесно связаны с дженериками, а их associated type обрабатывается похожим образом.

```
1  func min<T: Comparable>(_ x: T, _ y: T) -> T {
2      return y < x ? y : x
3  }
4  func min<T: Comparable>(_ x: TBox, _ y: TBox, valueWTable: VTable,
5                              comparableWTable: VTable) -> TBox {
6      let xCopy = valueWTable.copy(x)
7      let yCopy = valueWTable.copy(y)
8      let result = comparableWTable.lessThan(yCopy, xCopy) ? y : x
9                           valueWTable.release(xCopy) valueWTable.release(yCopy)
10     return result
11 }
```

## 5.9. Объединяем изученный материал

```swift
import Cocoa
import Contacts


protocol PhoneNumberProcotol {
    var number: String {get }
    var type: PhoneType {get }
}


enum PhoneType: String {
    case iPhone
    case mobile
    case home
    case other
}


struct PhoneNumber: PhoneNumberProcotol {
    var number: String
    var type: PhoneType
}


protocol AddressProtocol {
    var street: String {get }
    var city: String {get }
    var country: String {get }
}


struct Address: AddressProtocol {
    var street: String
    var city: String
    var country: String
}


```

```
protocol ContactProtocol {
    var firstName: String? {get }
    var lastName: String? {get }
    var phones: [PhoneNumberProcotol] {get }
    var addresses: [AddressProtocol] {get }
    var emails: [String] {get }
    var birthday: DateComponents? {get }
}


struct Contact: ContactProtocol {
    var firstName: String?
    var lastName: String?
    var phones: [PhoneNumberProcotol]
    var addresses: [AddressProtocol]
    var emails: [String]
    var birthday: DateComponents?
}


let calendar = Calendar.current
let now = Date()
var dateComponents = calendar.dateComponents([.year, .month, .day], from: now)


func generateContactWith(suffix: String, dateComponents: DateComponents?) -> Contact {
    return Contact(firstName: "firstName" + suffix, lastName: "lastName" + suffix,
        phones: [PhoneNumber(number: "+77777777777" + suffix, type: .iPhone)],
        addresses: [Address(street: "Street" + suffix, city: "
        ↪  City" + suffix, country: "Country" + suffix)],
        emails: ["firstName" + suffix + ".lastName" + suffix
                                                    + "@mail.com"],
        birthday:dateComponents)
}


let contact1 = generateContactWith(suffix: "1", dateComponents:dateComponents)

dateComponents.day = dateComponents.day! + 1
let contact2 = generateContactWith(suffix: "2", dateComponents:dateComponents)
```

```
78
79  let contact3 = generateContactWith(suffix: "3", dateComponents:dateComponents)
80
81  dateComponents.day = dateComponents.day! - 2
82  let contact4 = generateContactWith(suffix: "4", dateComponents:dateComponents)
83
84  dateComponents.day = 1
85  dateComponents.month = 1
86  let contact5 = generateContactWith(suffix: "5", dateComponents:dateComponents)
87
88  dateComponents.day = 31
89  dateComponents.month = 12
90  let contact6 = generateContactWith(suffix: "6", dateComponents:dateComponents)
91
92  let contact7 = generateContactWith(suffix: "7", dateComponents:nil)
93
94
95  typealias UpcomingBirthday = (Date, [ContactProtocol])
96  protocol ContactBookProtocol: RandomAccessCollection where
    ↪  Self.Element==ContactProtocol
97  {
98      mutating func add(_ contact: ContactProtocol)
99      func upcomingBirthdayContacts() -> [UpcomingBirthday]
100 }
101
102
103 struct ContactBook {
104     private var privateStorage = [ContactProtocol]()
105
106     enum SortingType {
107         case firstName
108         case lastName
109 }
110
111     var sortType = SortingType.firstName {
112         didSet {
113             sortPrivateStorage()
114     }
115 }
116
```

```
117     private mutating func sortPrivateStorage() {
118         privateStorage.sort(by: sortingClosure(for: sortType))
119     }
120 
121     private func sortingClosure(for sortingType: SortingType) ->
122                                 ((ContactProtocol, ContactProtocol) -> Bool) {
123 
124         switch sortingType {
125         case .firstName: return{type(of: self).isLessThan($0.firstName,$1.firstName)}
126         case .lastName: return{type(of: self).isLessThan($0.lastName, $1.lastName)}
127         }
128     }
129 
130     private static func isLessThan(_ lhs: String?, _ rhs: String?) -> Bool {
131         switch (lhs, rhs) {
132         case let (l?, r?):
133             return l.caseInsensitiveCompare(r) == .orderedAscending
134         case (nil, _?):
135             return true
136 
137         default:
138             return false
139         }
140     }
141 }
142 
143 
144 extension ContactBook: CustomDebugStringConvertible {
145     var debugDescription: String {
146         return self.reduce(into: "ContactBook:\n") {
147             result, anotherContact in
148 
149             let birthdayString: String
150             if let birthday = anotherContact.birthday {birthdayString =
                ↪  String(describing: birthday)
151             } else {
152                 birthdayString = ""
153             }
154 
155             if anotherContact.firstName == "" &amp;amp;&amp;
```

```
                                        anotherContact.lastName  == "" {
                    result += "\tunnamed contact \(birthdayString)\n"
                } else {
                    result += "\t\(anotherContact.firstName ?? "")
                    \(anotherContact.lastName ?? "") \(birthdayString)\n"
                }
            }
        }
    }


extension ContactBook: RandomAccessCollection {
    var startIndex: Int {
        return 0
    }

    var endIndex: Int {
        return privateStorage.count
    }

    func index(after i: Int) -> Int {
        return i + 1
    }

    subscript(position: Int) -> ContactProtocol {
        return privateStorage[position]
    }
}


extension Collection {
    func grouped&lt;Key: Equatable&gt;(by getKey:(_ element: Self.Element)
                                -> Key?) -> [(Key, [Self.Element])]{
        var result: [(Key, [Self.Element])] = []

    for element in self {
        guard let keyForElement = getKey(element) else {
            continue
        }

```

```
196             let index = result.index {
197                 (key: Key, elements: [Self.Element]) in
198 
199                 keyForElement == key
200             }
201 
202             if let index = index {
203                 result[index].1.append(element)
204             } else {
205                 result.append((keyForElement, [element]))
206             }
207         }
208         return result
209         }
210 }
211 
212 
213 extension ContactBook: ContactBookProtocol {
214     mutating func add(_ contact: ContactProtocol) {
215         let index = privateStorage.index {sortingClosure(for: sortType)(contact, $0) }
216 
217         privateStorage.insert(contact, at: index ?? privateStorage.endIndex)
218     }
219 
220     func upcomingBirthdayContacts() -> [UpcomingBirthday] {
221     let todayComponents = Calendar.current.dateComponents([.day, .month], from:
     ↪  Date())
222     guard let todayDayMonthDate = Calendar.current.date(from: todayComponents) else {
223         return [UpcomingBirthday]()
224     }
225 
226     let sortedBirthdays = grouped {
227         (anotherContact: ContactProtocol) -> Date? in
228 
229         guard var birthday = anotherContact.birthday else {
230             return nil
231         }
232 
233 
234         let dayMonthComponents = DateComponents(month:
```

```
                                    birthday.month, day: birthday.day)

                return Calendar.current.date(from: dayMonthComponents)
            }.sorted {
            $0.0 &lt; $1.0
            }

            let previousBirthdays = sortedBirthdays.prefix(while: {
             $0.0 &lt; todayDayMonthDate
            })

            let result = sortedBirthdays.dropFirst
                                        previousBirthdays.count) +previousBirthdays

            return [UpcomingBirthday](result.prefix(3))
        }
}


var contactBook = ContactBook()

contactBook.add(contact7)
contactBook.add(contact6)
contactBook.add(contact5)
contactBook.add(contact4)
contactBook.add(contact3)
contactBook.add(contact2)
contactBook.add(contact1)


print("\(contactBook)")


contactBook.count

for contact in contactBook {
// do something with contact
}
```

```
275
276 let authStatus = CNContactStore.authorizationStatus(for: .contacts)
277 if .notDetermined == authStatus {
278     let store = CNContactStore()
279     store.requestAccess(for: .contacts) {
280         (access, accessError) in
281
282
283         if access {
284             print("Access granted")
285         } else {
286             print("Access denied. Reason: \(String(describing:accessError))")
287         }
288     }
289 } else if .denied == authStatus {
290     print("Access has already denied")
291 } else {
292     print("Access has already granted")
293 }
294
295
296
297 struct CNPostalAddressWrapper: AddressProtocol {
298     private let address: CNLabeledValue&lt;CNPostalAddress&gt;
299
300     init(_ address: CNLabeledValue&lt;CNPostalAddress&gt;) {
301         self.address = address
302     }
303
304     var street: String {
305         return self.address.value.street
306     }
307
308     var city: String {
309         return self.address.value.city
310     }
311
312     var country: String {
313         return self.address.value.country
314     }
```

```
315     }
316
317
318 struct CNPhoneNumberWrapper: PhoneNumberProcotol {
319     private let phone: CNLabeledValue&lt;CNPhoneNumber&gt;
320
321     init(_ phone: CNLabeledValue&lt;CNPhoneNumber&gt;) {
322         self.phone = phone
323     }
324
325     var number: String {
326         return phone.value.stringValue
327     }
328
329     var type: PhoneType {
330         guard let label = self.phone.label else {
331             return .other
332         }
333
334         return PhoneType(rawValue: label) ?? .other
335     }
336 }
337
338
339 extension CNContact: ContactProtocol {
340     var firstName: String? {
341         return givenName
342     }
343
344     var lastName: String? {
345         return familyName
346     }
347
348     var phones: [PhoneNumberProcotol] {
349         return phoneNumbers.map(CNPhoneNumberWrapper.init)
350     }
351
352     var addresses: [AddressProtocol] {
353         return postalAddresses.map(CNPostalAddressWrapper.init)
354     }
```

```
355
356         var emails: [String] {
357             return emailAddresses.map{$0.value as String }
358         }
359     }
360
361
362     let store = CNContactStore()
363     let keys = [CNContactGivenNameKey,
364         CNContactFamilyNameKey,
365         CNContactPhoneNumbersKey,
366         CNContactEmailAddressesKey,
367         CNContactPostalAddressesKey,
368         CNContactBirthdayKey] as [CNKeyDescriptor]
369     let request = CNContactFetchRequest(keysToFetch: keys)
370
371
372     try? store.enumerateContacts(with: request) {
373         (contact: CNContact, _: UnsafeMutablePointer&lt;ObjCBool&gt;) in
374
375         contactBook.add(contact as ContactProtocol)
376     }
377
378
379     print("\(contactBook)")
380
381
382     let birthdays = contactBook.upcomingBirthdayContacts()
383
384
385     let dateFormatter = DateFormatter()
386     dateFormatter.dateFormat = "dd MMMM"
387
388
389     for (date, contacts) in birthdays {
390         print(dateFormatter.string(from: date))
391
392         for contact in contacts {
393             print("\t\(contact.firstName ?? "") \(contact.lastName ?? "")")
394         }
```

```
395 }
```

## 5.10. Decimal

Во время разработки приложения, вероятно, вам придется взаимодействовать с дробными числами: трансформировать их из строки, выполнять операции над ними и т.д. Для хранения чисел с плавающей точкой в Swift используются типы Float и Double. Однако, их точность не бесконечная и могут возникнуть проблемы. Особенно это важно если вы работаете с деньгами или другими величинами, требующими точное вычисление дробной части.

Рассмотрим простой пример. 100000 раз прибавим к сумме 0.01 и посмотрим на результат. Выполняется цикл не быстро, но самое плохое это то, что в итоге получается число 999.9999999992356, а не 1000 как мы ожидали.

```
1 var step = 0.01
2 var sum = 0.0
3 for _ in 1...100000 {
4     sum += step
5 }
6
7 sum // 999.9999999992356
```

Эта проблема не конкретного языка программирования, а реализации хранения обычных дробных чисел. Для ее решения создают специальный тип данных. В Swift это Decimal. Воспользуемся им вместо Double в нашем примере.

У Decimal есть много разных инициализаторов. Его можно создать и из целых чисел и из дробных литералов. Лучше всего воспользоваться конструктором, принимающим на вход строку. Его можно использовать и для инициализации строкой, введенной пользователем, и для инициализации данными, полученными от сервера. Если же ваш сервер присылает вам данные в виде float, то вам придется сначала считать число как Float или Double, а потом преобразовать в Decimal. Но в этом случае у вас уже может возникнуть ошибка из-за промежуточного шага. Поэтому лучше использовать для передачи дробных чисел строки и сразу их конвертировать в Decimal.

```
1 guard var decimalStep = Decimal(string: "0.01") else {
2     fatalError()
3 }
```

```
guard var decimalSum = Decimal(string: "0") else {
    fatalError()
}

for _ in 1...100000 {
    decimalSum += decimalStep
}

decimalSum // 1000
```

Помимо стандартных операций вроде сложения или умножения Decimal имеет много полезных функций. Рассмотрим некоторые из них.

```
guard var decimal = Decimal(string: "1003.07675") else {
    fatalError()
}

decimal.exponent // -5
decimal.significand // 1000307675
decimal.isInfinite // false

Decimal.pi // 3.14159265358979323846264338327950288419
Decimal.nan // NaN
```

Можно получить экспоненту или значимую часть числа, проверить является ли оно бесконечным. Также определено несколько статических констант, например число pi и нечисло NaN.

Математические операции можно выполнять с указанием способа округления. Оно будет применяться если полученное число нельзя сохранить без потерь. Всего есть четыре типа:

- plain округляет число в ближайшую сторону. Если число находится посередине, то округляет от нуля.
- down всегда округляет вниз.
- up всегда округляет вверх.
- bankers округляет в ближайшую сторону. Если число находится посередине, то округляет в ту сторону, где число получится четным.

| Начальное число | plain | down | up | bankers |
|---|---|---|---|---|
| 0.2 | 0 | 0 | 1 | 0 |
| 0.9 | 1 | 0 | 1 | 1 |
| 0.5 | 1 | 0 | 1 | 0 |
| 1.5 | 2 | 1 | 2 | 2 |
| -0.5 | 1 | -1 | 0 | 0 |

Для использования этой возможности придется выполнять вычисления через функции. Например:

```
var result = Decimal()
guard var leftDecimal = Decimal(string: "1.0") else {
    fatalError()
}

guard var rightDecimal = Decimal(string: "1003.07675") else {
    fatalError()
}

let error = NSDecimalAdd(&result, &leftDecimal, &rightDecimal, .plain)
if error != .noError {
    fatalError()
}

result // 10004.07675
```

Как видите, лишнего кода получается много. Нужно передать оба операнда и переменную для сохранения результата. Причем их нужно передать через указатели. На вызывающей стороне это выглядит также как inout параметр.

Если же вы хотите просто выполнить округление, воспользуйтесь функцией NSDecimalRound(). В нее тоже нужно передать тип округления и количество знаков после запятой.

```
var result = Decimal()

guard var decimal = Decimal(string: "1003.07675") else {
```

```
    fatalError()
}

NSDecimalRound(&result, &decimal, 2, .bankers)

result // 10003.08
```

С полной информацией о Decimal вы можете ознакомится в документации. Обратите внимание, что есть еще Objective-C класс NSDecimalNumber. Взаимодействие между языками получилось немного запутанным. В Objective-C есть и класс NSDecimalNumber. и структура NSDecimal. Плюс NSDecimalNumber в том, что его можно использовать с NSNumberFormatter(о нем чуть позже) т.к. он ожидает на вход любой класс. id - это указатель на любой экземпляр класса.

```
- (nullable NSString *)stringForObjectValue:(nullable id)obj;
```

В Swift NSDecimalNumber импортируется как структура Decimal. Но благодаря тому, что начиная с Swift 3 id импортируется как Any, а не AnyObject, вы можете использовать Decimal с NumberFormatter.

```
open func string(for obj: Any?) -> String?
```

Поэтому, если у вас в приложении нет взаимодействия Objective-C и Swift, то используйте Decimal и не думайте о существовании NSDecimalNumber. Но если вам нужно использовать оба языка, то будьте осторожны. Decimal из Swift импортируется в Objective-C уже не как NSDecimalNumber, а как NSDecimal.

Пара слов о NumberFormatter. Денежные суммы как и даты и нельзя выводить вручную. В разных странах они отображаются по разному. Для решения этой проблемы у нас есть различные форматтеры. NumberFormatter используется для вывода денежных сумм, обычных чисел с плавающей запятой, процентов, порядковых чисел и других данных.

В примере ниже создается NumberFormatter и ему указывается стиль currency. decimal будет выводиться как сумма в долларах т.к. по умолчанию используются настройки системы.

```
guard var decimal = Decimal(string: "1003.07675") else {
    fatalError()
}

let formatter = NumberFormatter()
formatter.numberStyle = .currency
formatter.string(for: decimal) // "$1,003.08"
```

Можно указать другую локализацию с помощью структуры Locale. Теперь будет использоваться другой символ валюты, другой разделитель целой части и разделитель тысяч.

```
guard var decimal = Decimal(string: "1003.07675") else {
    fatalError()
}

let formatter = NumberFormatter()
formatter.numberStyle = .currency

formatter.locale = Locale(identifier: "ru_RU")
formatter.string(for: decimal) // "1 003,08 "
```

Это была лишь верхушка большой темы - интернационализации. Подробнее об этом мы еще поговорим в следующих курсах. О всех возможностях NumberFormatter вы можете прочесть в документации, а пока запомните, что нельзя выводить даты и другие языкозависимые данные как есть или вручную пытаться их форматировать. Даже если вы считаете, что в вашем приложении будет поддерживаться только один язык.

**О проекте**
Академия e-Legion — это образовательная платформа для повышения квалификации в мобильной разработке. Слушайте лекции топовых разработчиков, выполняйте практические задания и прокачивайте свои скиллы. Получите высокооплачиваемую профессию — разработчик мобильных приложений.

## Программа “iOS-разработчик”

### Блок 1. Введение в разработку Swift

- Знакомство со средой разработки Xcode
- Основы Swift
- Обобщённое программирование, замыкания и другие продвинутые возможности языка

### Блок 2. Пользовательский интерфейс

- Особенности разработки приложений под iOS
- UIView и UIViewController
- Создание адаптивного интерфейса
- Анимации и переходы
- Основы отладки приложений

### Блок 3. Многопоточность

- Способы организации многопоточности
- Синхронизация потоков
- Управление памятью
- Основы оптимизации приложений

### Блок 4. Работа с сетью

- Использование сторонних библиотек
- Основы сетевого взаимодействия
- Сокеты
- Парсинг данных

- Основы безопасности

### Блок 5. Хранение данных

- Способы хранения данных
- Core Data
- Accessibility

### Блок 6. Мультимедиа и другие фреймворки

- Работа с аудио и видео
- Интернационализация и локализация
- Геолокация
- Уведомления
- Тестирование приложений