



Урок 7

Обработка ошибок и исключений.

Учимся искать ошибки и предсказывать их появление.
Исключения try/catch. Error.

[Error](#)

[nil](#)

[Guard](#)

[Error](#)

[Try/Catch](#)

[Throws](#)

[Обработка ошибок](#)

[Домашнее задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

Error

nil

В вашей программе часто происходят ошибки или просто ошибочные ситуации. Вы хотели найти среднюю заработную плату у работников, а работников не оказалось. Вы хотели найти конкретного работника, а его не оказалось, хотя он там должен был быть. Отправили запрос на сервер, а ответа не пришло. Все эти ситуации ошибочны.

Давайте напишем пример. Класс "Factory" олицетворяет наше предприятие. Словарь - это зарплатная ведомость. Метод "salary" возвращает зарплату для одного работника.

```
// Наша фабрика
class Factory {
    // зарплата рабочих
    fileprivate var employee = [
        "Иванов Иван Иванович": 3000,
        "Сидоров Сидор Сидорович": 12080,
        "Петров Петр Петрович": 8040
    ]
    // получаем зарплату рабочего
    func salary(atFio fio: String) -> Int {
        // так как такого рабочего может и не быть, разворачиваем опциональную переменную
        return employee[fio]!
    }
}
```

Словарь всегда возвращает опциональное значение, а наш метод нет, поэтому мы воспользовались принудительным разворачиванием опционального значения. На самом деле это плохая практика, если мы передадим ФИО и его не окажется в словаре, то наша программа упадет. Чтобы этого избежать можно добавить значение по умолчанию, например, мы можем вернуть 0, если не найдем сотрудника.

```
class Factory {
    fileprivate var employee = [
        "Иванов Иван Иванович": 3000,
        "Сидоров Сидор Сидорович": 12080,
        "Петров Петр Петрович": 8040
    ]
    func salary(atFio fio: String) -> Int {
        // если рабочего нет, вернем 0
        return employee[fio] ?? 0
    }
}
```

Теперь наша программа не упадет, но есть минус. Мы можешь запросить зарплату для рабочего, которого нет и получим 0, можно легко решить, что у него был отпуск без содержания или просто

штраф. Программа будет работать, но она может работать неправильно, а мы и не узнаем. Самый верный способ решить эту проблему, сделать возвращаемый результат тоже опциональным.

```
class Factory {
  private var employee = [
    "Иванов Иван Иванович": 3000,
    "Сидоров Сидор Сидорович": 12080,
    "Петров Петр Петрович": 8040
  ]
  // теперь наш метод возвращает опциональное значение
  func salary(atFio fio: String) -> Int? {
    return employee[fio]
  }
}

let factory = Factory()
factory.salary(atFio: "Иванов Иван Иванович")// 3000
factory.salary(atFio: "Брюс Вейн Бэтманович")// nil
```

Таким образом при вызове функции мы точно будем знать, что не ошиблись в написании ФИО или у нас не закрался фантомный работник.

Guard

Давайте добавим метод подсчета средней зарплаты по предприятию.

```
class Factory {
  private var employees = [
    "Иванов Иван Иванович": 3000,
    "Сидоров Сидор Сидорович": 12080,
    "Петров Петр Петрович": 8040
  ]
  // теперь наш метод возвращает опциональное значение
  func salary(atFio fio: String) -> Int? {
    return employees[fio]
  }
  func averageSalary() -> Int {
    var totalSalary = 0
    for employee in employees {
      totalSalary += employee.value
    }
    return totalSalary / employees.count
  }
}

let factory = Factory()
factory.salary(atFio: "Иванов Иван Иванович")// 3000
factory.salary(atFio: "Брюс Вейн Бэтманович")// nil
factory.averageSalary() // 7706
```

Выглядит отлично, средняя зарплата посчитана. Но есть один неочевидный нюанс, если у нас не будет ни одного сотрудника, то мы получим деление на 0 и программа упадет. Это очень

распространенная ошибка, так как обычно программист не думает, что кто-то будет рассчитывать зарплату для предприятия, на котором нет сотрудников, но опыт говорит об обратном, если ваша программа может упасть, то пользователь обязательно ее уронит.

Давайте добавим проверку на деление на 0.

```
func averageSalary() -> Int {  
    // если на предприятии есть рабочие, то считаем среднюю зарплату  
    if employees.count > 1 {  
        var totalSalary = 0  
        for employee in employees {  
            totalSalary += employee.value  
        }  
        return totalSalary / employees.count  
    } else { // если нет работников, то возвращаем 0  
        return 0  
    }  
}
```

Теперь мы защищены от ошибки, а пользователь увидит то, что ожидает, если нет работников, то их средняя зарплата равна нулю.

Раньше мы упоминали ключевое слово “guard”, но у нас не было с ним примеров. Это отличный шанс им воспользоваться, чтобы сделать код лаконичнее.

```
func averageSalary() -> Int {  
    // нам необходимо, чтобы на предприятии работали сотрудники  
    guard employees.count > 1 else { // иначе возвращаем 0  
        return 0  
    }  
    // если наше требование удовлетворено, то просто переходим дальше к выполнению метода  
    var totalSalary = 0  
    for employee in employees {  
        totalSalary += employee.value  
    }  
    return totalSalary / employees.count  
}
```

Посмотрите, что изменилось. Мы заменили “if” на “guard” и вынесли код из вложенного блока. Guard очень похож на if, но имеет другой синтаксис, а самое главное - семантическое отличие. Его часто называют охранным оператором, так как он охраняет ваш код от выполнения в тех случаях, когда он выполняться не должен.

Он проверяет условие также как и “if”, но только он, если условие верно, не выполняет никаких блоков, он просто переходит к основному коду, объявленному за ним. Но вот если условие не верно, то выполняется блок “else”, в котором вы можете произвести какие-либо действия для обработки сложившейся ситуации, но самое главное, вы должны покинуть область видимости, например, если вы находитесь в методе, то вы обязаны сделать “return”. Такой синтаксис делает вашу программу намного читабельнее.

Error

Давайте напишем еще один пример, его рекомендуют сами apple. Это простой вендинговый автомат, у нас есть продукты, позиции в наличии и само хранилище в автомате. Покупатель бросает монетки и заказывает себе товар. Но что, если продукта нет в наличии, или он вообще продается в нашем автомате или недостаточно денег? Тогда мы по аналогии с прошлым примером вернем nil.

```

// позиции в автомате
struct Item {
    var price: Int
    var count: Int
    let product: Product
}

// товар
struct Product{
    let name: String
}

// вендинговая машина
class VendingMachine {
// хранилище
    var inventory = [
        "Candy Bar": Item(price: 12, count: 7, product: Product(name: "Candy Bar")),
        "Chips": Item(price: 10, count: 4, product: Product(name: "Chips")),
        "Pretzels": Item(price: 0, count: 11, product: Product(name: "Pretzels"))
    ]
// количество денег, переданным покупателем
    var coinsDeposited = 0
// продаем товар
    func vend(itemNamed name: String) -> Product? {
// если наша машина не знает такого товара вообще, возвращаем nil
        guard let item = inventory[name] else {
            return nil
        }
// если товара нет в наличии, возвращаем nil
        guard item.count > 0 else {
            return nil
        }
// если недостаточно денег, возвращаем nil
        guard item.price <= coinsDeposited else {
            return nil
        }
// продаем товар
        coinsDeposited -= item.price
        var newItem = item
        newItem.count -= 1
        inventory[name] = newItem
        print("Dispensing \(name)")
        return newItem.product
    }
}

let vendingMachine = VendingMachine()
vendingMachine.vend(itemNamed: "Snikers") // nil
vendingMachine.vend(itemNamed: "Candy Bar")// nil
vendingMachine.vend(itemNamed: "Pretzels") // nil

```

Присмотритесь внимательно, мы попытались купить несколько товаров, но все попытки завершились неудачей. Каких-то товаров нет в ассортименте, других в наличии, и денег мы не положили, но мы

понятия не имеем о том, что именно сделали нет, ведь нам всегда возвращается nil без каких-либо пояснений. Вариант обработки ошибок с возвращением nil не очень хорошо подходит для случаев, когда нам необходимо не только получить ошибку, но и пояснение, что же именно пошло не так.

Для таких случаев у нас есть специальный протокол "Error". Он применяется к перечислениям, превращая их в перечисление вариантов ошибок. Например, ошибки для нашего автомата будут выглядеть так.

```
enum VendingMachineError: Error {           // ошибки автомата

    case invalidSelection                    // нет в ассортименте
    case outOfStock                         // нет в наличии
    case insufficientFunds(coinsNeeded: Int) // недостаточно денег, передаем недостаточную сумму
}
```

Теперь перепишем наш пример. Во-первых, наша функция продажи будет возвращать кортеж из продукта и ошибки. Оба значения опциональные, ведь если есть ошибка, нет продукта и наоборот. После покупки товара мы проверим наличие продукта, если его нет, то проверим ошибку, чтобы выяснить, что пошло не так.

```
// Вендинговая машина
class VendingMachine {
    // Хранилище
    var inventory = [
        "Candy Bar": Item(price: 12, count: 7, product: Product(name: "Candy Bar")),
        "Chips": Item(price: 10, count: 4, product: Product(name: "Chips")),
        "Pretzels": Item(price: 0, count: 11, product: Product(name: "Pretzels"))
    ]
    // Количество денег переданным покупателем
    var coinsDeposited = 0
    // Продаем товар
    func vend(itemNamed name: String) -> (Product?, VendingMachineError?) { // возвращаем кортеж из
        товара и ошибки
        // Если наша машина не знает такого товара вообще
        guard let item = inventory[name] else {
            // Возвращаем nil вместо продукта и ошибку
            return (nil, VendingMachineError.invalidSelection)
        }
        // Если товара нет в наличии
        guard item.count > 0 else {
            // возвращаем nil вместо продукта и ошибку
            return (nil, VendingMachineError.outOfStock)
        }
        // Если недостаточно денег, возвращаем nil
        guard item.price <= coinsDeposited else {
            // возвращаем nil место продукта и ошибку
            return (nil, VendingMachineError.insufficientFunds(coinsNeeded: item.price - coinsDeposited))
        }
        // продаем товар
        coinsDeposited -= item.price
        var newItem = item
    }
}
```

```

        newItem.count -= 1
        inventory[name] = newItem
// Возвращаем nil вместо ошибки и продукт
        return (newItem.product, nil)
    }
}
let vendingMachine = VendingMachine()
let sell1 = vendingMachine.vend(itemNamed: "Snickers") // nil, invalidSelection
let sell2 = vendingMachine.vend(itemNamed: "Candy Bar") // nil, (insufficientFunds, 12)
let sell3 = vendingMachine.vend(itemNamed: "Pretzels") // Product("Pretzels"), nil
if let product = sell1.0 {
    print("Мы купили: \(product.name)")
} else if let error = sell1.1 {
    print("Произошла ошибка: \(error.localizedDescription)")
}

```

Теперь, когда мы совершаем покупку и происходит ошибка, мы получаем подробное описание ошибки.

Try/Catch

Throws

Но в нашем примере все же есть недочеты. Во-первых, может произойти так, что мы не вернем ни ошибки ни продукта. Во-вторых, мы при вызове метода покупки мы не видим явно, что он может вызывать ошибки. Да и возвращать такие кортежи не очень удобно. Специально для таких случаев у нас есть ключевое слово “throws”. Если пометить метод этим ключевым словом, то мы однозначно покажем, что он может содержать ошибку и ее надо обработать, мы просто не оставим пространства для маневра тому, что вызовет наш метод.

Все методы, переменные, как “throws”, должны возвращать исключение. Исключение - это наши ошибки, имплементирующие протокол “Error”, но не просто возвращаемые, а сгенерированные в исключительную ситуацию. Чтобы сгенерировать ошибку, необходимо использовать ключевое слово “throw”.

```

// Вендинговая машина
class VendingMachine {
// Хранилище
    var inventory = [
        "Candy Bar": Item(price: 12, count: 7, product: Product(name: "Candy Bar")),
        "Chips": Item(price: 10, count: 4, product: Product(name: "Chips")),
        "Pretzels": Item(price: 0, count: 11, product: Product(name: "Pretzels"))
    ]
// Количество денег, переданным покупателем
    var coinsDeposited = 0
// продаем товар
// возвращаем продукт,
// но помечаем метод как "throws", это означает, что он может завершиться с ошибкой
    func vend(itemNamed name: String) throws -> Product {

```



```

// Если наша машина не знает такого товара вообще,
guard let item = inventory[name] else {
// генерируем ошибку
    throw VendingMachineError.invalidSelection
}
// если товара нет в наличии,
guard item.count > 0 else {
// генерируем ошибку
    throw VendingMachineError.outOfStock
}
// если недостаточно денег,
guard item.price <= coinsDeposited else {
// генерируем ошибку
    throw VendingMachineError.insufficientFunds(coinsNeeded: item.price - coinsDeposited)
}
// продаем товар
coinsDeposited -= item.price
var newItem = item
newItem.count -= 1
inventory[name] = newItem
// возвращаем продукт
return newItem.product
}
}

```

Обратите внимание, что продукт у нас не опционального типа, а обычного, но также указано ключевое слово “throws”. Это означает, что наш метод всегда возвращает продукт, но во время этого выполнения может произойти ошибка и ее необходимо обработать.

Обработка ошибок

Нельзя просто взять и использовать результат метода, генерирующего исключения. Во-первых, перед его вызовом необходимо поставить ключевое слово “try”, что дословно переводится как попытка. Мы, фактически, пытаемся выполнить метод, но у нас может и не получиться и вместо результата мы получим ошибку. Так как метод у нас “опасный”, мы должны поместить его вызов в специальный блок “do/catch”.

```

do {
    // помечаем метод как try и помещаем его вызов в блок do
    let sell1 = try vendingMachine.vend(itemNamed: "Snikers")
} catch let error {
    // если во время выполнения возникла ошибка, обрабатываем ее
    print(error.localizedDescription)
}

```

Внутри блока “do” мы безопасно вызываем метод, а если при этом произойдет ошибка, то мы тут же перейдем к блоку “catch” и обработаем ее, например, выведем в консоль.

У блока “catch” есть особенность, он работает как “switch” и позволяет по-разному обрабатывать разные ошибки.

```
do {
  let sell1 = try vendingMachine.vend(itemNamed: "Snickers")
} catch VendingMachineError.invalidSelection {
  print("Такого товара не существует")
} catch VendingMachineError.insufficientFunds(let coinsNeeded) {
  print("Введенная сумма недостаточна. Необходимо еще \(coinsNeeded) монет")
}
```

На самом деле, если вам в момент вызова не важно какая ошибка возникает, вы можете на ходу преобразовать результат в опциональное значение. При этом можно не писать блоки “do/catch” а просто добавить “?” после слова “try”. Вы также можете использовать “!”, но так делать никогда не рекомендуется, так как мы вернемся к тому, с чего начали, и наша программа будет падать при ошибке.

```
// добавляем ? после try и результат становится опциональным, в случае ошибки получаем nil
let sell = try? vendingMachine.vend(itemNamed: "Snickers")
```

Последняя особенность генерации исключений состоит в том, что если “throw” методы вызываются один из другого, вы можете не обрабатывать ошибку в каждом из них, а передавать ее наверх по цепочке. Давайте добавим в наш пример покупателей и метод покупки из автомата. Этот метод будет искать покупателя в списке, в случае, если он его найдет, попытается купить его, продукт, который ему нужен. В случае, если покупателя не будет найдено, мы генерируем новую ошибку.

```
enum BuyError: Error { // ошибки покупки
  case buyrNotFound // покупатель не найден
}
// список покупателей и товаров, которые они хотят купить
let favoriteSnacks = [
  "Alice": "Chips",
  "Bob": "Licorice",
  "Eve": "Pretzels",
]
// метод покупки, тоже генерирует исключение
func buyFavoriteSnack(person: String, vendingMachine: VendingMachine) throws -> Product {
  // если покупатель не найден, генерируем исключение
  guard let snackName = favoriteSnacks[person] else {
    throw BuyError.buyrNotFound
  }
  // иначе покупаем товар
  return try vendingMachine.vend(itemNamed: snackName)
}
let vendingMachine = VendingMachine()
do {
  let sell = try buyFavoriteSnack(person: "Alice", vendingMachine: vendingMachine)
} catch VendingMachineError.invalidSelection {
  print("Такого товара не существует")
} catch VendingMachineError.insufficientFunds(let coinsNeeded) {
  print("введенная сумма недостаточна. Необходимо еще \(coinsNeeded) монет")
} catch BuyError.buyrNotFound {
  print("Покупатель не найден")
}
```

```
}
```

Обратите внимание, мы не обрабатываем исключение внутри метода “buyFavoriteSnack”, так как он тоже генерирует исключения. Фактически, он просто вернет в результате исключения метода “vend”. А уже при вызове метода покупки мы обработаем все возможные исключения сразу.

Механизм очень мощный и полезный инструмент, который не следует игнорировать при разработке программы. Случаются ситуации, в которых верно добавленное исключение поможет сэкономить часы отладки.

Домашнее задание

1. Придумать класс методы, которого могут создавать непоправимые ошибки. Реализовать их с помощью try/catch.
2. Придумать класс методы которого могут завершаться неудачей. Реализовать их с использованием Error.

Дополнительные материалы

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

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

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