Лабораторная работа 10

ОПЕРАТОРЫ, ОБРАБОТКА ОШИБОК

Обработка ошибок

В Swift ошибки отображаются значениями типов, которые соответствуют протоколу Error. Этот пустой протокол является индикатором того, что это перечисление может быть использовано для обработки ошибок.

В Swift для создания перечня возможных ошибок служат перечисления, где каждый член перечисления соответствует отдельной ошибке.

enum VendingMachineError: Error {

case InvalidSelection

case InsufficientFunds(coinsNeeded: Int)

case OutOfStock

}

Процесс появления ошибки называется выбрасыванием ошибки. Для того чтобы выбросить ошибку, необходимо воспользоваться оператором throw.

throw VendingMachineError.InsufficientFunds(coinsNeeded: 5)

Сам по себе выброс ошибки не приносит каких-либо результатов. Выброшенную ошибку необходимо перехватить и корректно обработать. В Swift существует четыре способа обработки ошибок:

* передача ошибки;
* обработка ошибки оператором do-catch;
* преобразование ошибки в опционал;
* запрет на выброс ошибк.

Если при вызове какой-либо функции или метода вы знаете, что он может выбросить ошибку, то необходимо перед вызовом указывать ключевое слово try. Теперь разберем каждый из способов обработки ошибок.

Передача ошибки

При передаче ошибки блок кода (функция, метод или инициализатор), ставший источником ошибки, самостоятельно не обрабатывает ее, а передает выше в код, который вызвал данный блок кода. Для того чтобы указать блоку кода, что он должен передавать возникающие в нем ошибки, в реализации данного блока после списка параметров указывается ключевое слово throws. Ниже приведен пример объявления двух функций, которые передают возникающие в них ошибки выше.

func anotherFunc() throws {

// тело функции

var value = try someFunc()

// ...

}

func someFunc() throws -> String{

// тело функции

try anotherFunc()

// ...

}

try someFunc();

Функция someFunc() возвращает значение типа String, поэтому ключевое слово throws указывается перед типом возвращаемого значения. Функция anotherFunc() в своем теле самостоятельно не выбрасывает ошибки, она может лишь перехватить ошибку, выброшенную функцией anotherFunc(). Для того чтобы перехватить ошибку, выброшенную внутри блока кода, необходимо осуществлять вызов с помощью упомянутого ранее оператора try. Благодаря ему функция anotherFunc() сможет отреагировать на возникшую ошибку так, будто она сама является ее источником. А так как эта функция также помечена ключевым словом throws, она просто передаст ошибку в вызвавший ее код. Если функция не помечена ключевым словом throw, то все возникающие внутри нее ошибки она должна обрабатывать самостоятельно.

Функция, отмеченная throws называется генерирующей функцией.

Оператор do-catch

Выброс и передача ошибок вверх в конце концов должна вести к их обработке таким образом, чтобы это принесло определенную пользу пользователю и разработчику. Для этого вы можете задействовать оператор do-catch.

СИНТАКСИС

do {

try имяВызываемогоБлока

} catch шаблон1 {

// код...

} catch шаблон2 {

// код...

}

Оператор содержит блок do и произвольное количеств облоков catch. В блоке do должен содержаться вызов функции или метода, которые могут выбросить ошибку. Вызов осуществляется с помощью оператора try. Если в результате вызова была выброшена ошибка, то данная ошибка сравнивается с шаблонами в блоках catch. Если в одном из них найдено совпадение, то выполняется код из данного блока. Вы можете использовать ключевое слово where в шаблонах условий. Блок catch можно задействовать без указания шаблона. В этом случае данный блок соответствует любой ошибке, а сама ошибка будет находиться в локальной переменной error.

Используем оператор do-catch, чтобы перехватить и обработать возможные ошибки

var vendingMachine = VendingMachine()

vendingMachine.coinsDeposited = 8

do {

try buyFavoriteSnack(person: "Alice", vendingMachine: vendingMachine)

} catch VendingMachineError.InvalidSelection {

print("Invalid Selection.")

} catch VendingMachineError.OutOfStock {

print("Out of Stock.")

} catch VendingMachineError.InsufficientFunds(let coinsNeeded) {

print("Недостаточно средств. Пожалуйста, внесите еще \ (coinsNeeded) монет(ы).")

}

// выводит "Недостаточно средств. Пожалуйста, внесите еще 2 монет(ы)."

Преобразование ошибки в опционал

Для преобразования выброшенной ошибки в опциональное значение используется оператор try, а точнее, его форма try?. Если в этом случае выбрасывается ошибка, то значение выражения вычисляется как nil.

func someThrowingFunction() throws -> Int {

// ...

}

let x = try? someThrowingFunction()

Если функция someThrowingFunction() выбросит ошибку, то в константе x окажется значение nil.

Запрет на выброс ошибки

В некоторых ситуациях можно быть уверенным, что блок кода во время исполнения не выбросит ошибку. В этом случае необходимо использовать оператор try!, который сообщает о том, что данный блок гарантированно не выбросит ошибку, — это запрещает передачу ошибки в целом .

let photo = try! loadImage("./Resources/John Appleseed.jpg”)

Функция loadImage(\_:) производит загрузку локального изображения, а в случае его отсутствия выбрасывает ошибку. Так как указанное в ней изображение является частью разрабатываемой вами программы и гарантированно находится по указанному адресу, с помощью оператора try! целесообразно отключить режим передачи ошибки. Будьте внимательны: если при запрете передачи ошибки блок кода все же выбросит ее, то ваша программа экстренно завершится.

**Отложенные действия по очистке**

Swift позволяет определить блок кода, который будет выполнен лишь по завершении выполнения текущей части программы. Для этого служит оператор defer, который содержит набор отложенных выражений . С его помощью вы можете выполнить необходимую очистку независимо от того, как произойдет выход из данной части программы. Отложенные действия выполняются в обратном порядке, то есть вначале выполняется блок последнего оператора defer, затем предпоследнего и т . д . Рассмотрим пример использования блока отложенных действий

if exists(filename) {

let file = open(filename)

defer {

close(file)

}

while let line = try file.readline() {

// работа с файлом.

}

}

}

В данном примере оператор defer просто обеспечивает закрытие открытого ранее файла.

Вы можете использовать оператор defer, даже если не используете кода обработки ошибок.

**Использование операторов**

Возможна ситуация, в которой для ваших собственных объектных типов существующие операторы окажутся бесполезными. В таком случае вам потребуется самостоятельно создать свои реализации стандартных операторов или полностью новые операторы.

**Инфиксные, префиксные и постфиксные операторы**

Оператор сложения является бинарным инфиксным, то есть он ставится между двумя операндами. Помимо инфиксных операторов, в Swift существуют префиксные (предшествуют операнду) и постфиксные (следуют за операндом) операторы. Для перегрузки префиксного или постфиксного оператора перед объявлением операторной функции необходимо указать модификатор prefix или postfix соответственно.

struct Vector2D {

var x = 0.0, y = 0.0

}

extension Vector2D {

static func + (left: Vector2D, right: Vector2D) -> Vector2D {

return Vector2D(x: left.x + right.x, y: left.y + right.y)

}

}

let vector = Vector2D(x: 3.0, y: 1.0)

let anotherVector = Vector2D(x: 2.0, y: 4.0)

let combinedVector = vector + anotherVector

// (5.0, 5.0)

extension Vector2D

{

static prefix func - (vector: Vector2D) -> Vector2D {

return Vector2D(x: -vector.x, y: -vector.y)

}

}

Составной оператор присваивания

Для перегрузки составных операторов в операторной функции первый передаваемый аргумент необходимо сделать сквозным (inout), так как именно его значение будет меняться в ходе выполнения функции . В примере пример реализации составного оператора присваивания-сложения для экземпляров типа Vector2D.

static func += ( left: inout Vector2D, right: Vector2D) {

left = left + right

}

var original = Vector2D(x: 1.0, y: 2.0)

let vectorToAdd = Vector2D(x: 3.0, y: 4.0)

original += vectorToAdd

// original теперь имеет значения (4.0, 6.0)

Оператор эквивалентности

Пользовательские объектные типы не содержат встроенной реализации оператора эквивалентности, поэтому чтобы сравнить два экземпляра, необходимо перегрузить данный оператор с помощью операторной функции .

static func == (left: Vector2D, right: Vector2D) -> Bool {

return (left.x == right.x) && (left.y == right.y)

}

static func != (left: Vector2D, right: Vector2D) -> Bool {

return !(left == right)

}

let twoThree = Vector2D(x: 2.0, y: 3.0)

let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)

if twoThree == anotherTwoThree {

print("Эти два вектора эквивалентны.")

}

// выводит "Эти два вектора эквивалентны."

В операторной функции == мы реализуем всю логику сравнения двух экземпляров типа Vector2D. Так как данная функция возвращает false в случае неэквивалентности операторов, мы можем использовать ее внутри собственной реализации оператора неэквивалентности.

Пользовательские операторы

В дополнение к стандартным операторам языка Swift вы можете определять собственные. Собственные операторы объявляются с помощью ключевого слова operator и модификаторов prefix, infix и postfix, причем вначале необходимо объявить новый оператор, а уже потом задавать его новую реализацию в виде операторной функции. В следующем примере реализуется новый оператор +++, который складывает экземпляр типа Vector2D сам с собой.

prefix operator +++

extension Vector2D {

prefix func +++ (vector: inout Vector2D) -> Vector2D

{ vector += vector

return vector

}

}

var toBeDoubled = Vector2D(x: 1.0, y: 4.0)

let afterDoubling = +++toBeDoubled

// toBeDoubled теперь имеет значения (1.0, 4.0)

// afterDoubling также имеет значения (2.0, 8.0)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Задание 10.1**

1. *Добавьте для любого из классов, определенных ранее или структуры новый пользовательский оператор.*
2. *Продемонстрируйте понимание обработки ошибок написав обработчик любой ситуации наподобие примера из теоритической части. Обработчик должен продемонстрировать различные результаты, как успешные, так и ошибочные*

ВОПРОСЫ:

1. Как определить и выбросить ошибку?
2. Какие есть способы обработки ошибок?
3. Как выполнить передачу ошибки из блока кода?
4. Как использовать оператор do-catch? Можно задасть catch без шаблона?
5. Как преобразовать ошибку в опционал?
6. Как сделать запрет на выброс ошибки?
7. Зачем и как используется опреатор defer?
8. Приведите пример определния оператора для типа (оперделение стандартного и нового)?