**Лабораторная работа 9**

**РАЗГРАНИЧЕНИЕ ДОСТУПА, УНИВЕРСАЛЬНЫЕ ШАБЛОНЫ**

**Разграничение доступа**

Уровни доступа

База системы разграничения доступа при разработке приложений строится на основе понятия «модуль». Всего Swift предлагает пять различных уровней доступа для объектов вашего кода:

* open и public Открытый и публичный. Данные уровни доступа очень похожи, они открывают полную свободу использования объекта. Вы можете импортировать модуль и свободно использовать его public-объекты в своем коде.
* internal Внутренний. Данный уровень используется в случаях, когда необходимо ограничить использование объекта самим модулем. Таким образом, объект будет доступен во всех исходных файлах модуля, исключая его использование за пределами модуля.
* fileprivate Частный в пределах файла. Данный уровень позволяет использовать объект только в пределах данного исходного файла.
* private Частный. Данный уровень позволяет использовать объект только в пределах конструкции, в которой он объявлен. Например, объявленный в классе параметр не будет доступен в его расширениях.

Открытый уровень доступа является самым высоким и наименее ограничивающим, а частный — самым низким и максимально ограничивающим. Открытый уровень применяется только к классам и членам класса (свойствам, методам и т . д .) и отличается от публичного следующими характеристиками:

* Класс, имеющий уровень доступа public (или более строгий), может *иметь подклассы* только в том модуле, где он был объявлен.
* Члены класса, имеющие уровень доступа public (или более строгий), могут быть *переопределены* (с помощью оператора override) в подклассе только в том модуле, где он объявлен.
* Класс, имеющий уровень доступа open, может *иметь подклассы* внутри модуля, где он определен, и в модулях, импортированных в данном модуле.
* Члены класса, имеющие уровень доступа open, могут быть *переопределены* (с помощью оператора override) в подклассе в том модуле, где он объявлен, а также в модулях, импортируемых в данном модуле.

Главное правило определения уровня доступа в Swift звучит следующим образом: объект с более низким уровнем доступа не может определить объект с более высоким уровнем доступа (в контексте модуля). *По умолчанию все объекты вашего кода имеют уровень доступа* internal. Для того чтобы изменить его, необходимо явно указать уровень. При этом если вы разрабатываете фреймворк, то для того, чтобы сделать некоторый объект частью доступного API, вам необходимо изменить его уровень доступа на public.

Чтобы определить уровень доступа к некоторому объекту, необходимо указать соответствующее ключевое слово (open, public, internal, private) перед определением объекта (func, property, class, struct и т.д .).

open class SomePublicClass {}

internal class SomeInternalClass {}

fileprivate class SomePrivateClass {}

public var somePublicVar = 0

private var somePrivatelet = 0

internal func someInternalFunc() {}

Уровень доступа к типам данных

Swift позволяет определять собственные типы данных. Если вам требуется указать уровень доступа к типу данных или его членам, то это необходимо сделать в момент определения типа. Новый тип данных может быть использован там, где это позволяет его уровень доступа. Если ваш объект имеет вложенные объекты (например, класс со свойствами и методами), то *уровень доступа родителя определяет уровни доступа к его членам*. Таким образом, если вы укажете уровень доступа private, то все его члены по умолчанию будут иметь уровень доступа private. Для уровней доступа public и internal уровень доступа членов — internal.

public class SomePublicClass { // public класс

public var somePublicProperty = 0 // public свойство

var someInternalProperty = 0 // internal свойство

fileprivate func somePrivateMethod() {} // fileprivate метод

}

class SomeInternalClass { // internal класс

var someInternalProperty = 0 // internal свойство

private func somePrivatemethod() {} // private метод

}

private class SomePrivateClass { // private класс

var somePrivateProperty = 0 // private свойство

func somePrivateMethod() {} // private метод

}

При наследовании уровень доступа подкласса не может быть выше уровня родительского класса. Уровень доступа к кортежу типа данных определяется наиболее строгим типом данных, включенным в кортеж.

Уровень доступа к функции определяется самым строгим уровнем типов аргументов функции и типа возвращаемого значения.

func someFunction() -> (SomeInternalClass, SomePrivateClass) {

// тело функции

}

Можно было ожидать, что уровень доступа функции будет равен internal, так как не указан явно. На самом деле эта функция вообще не будет скомпилирована. Это связано с тем, что тип возвращаемого значения — это кортеж с уровнем доступа private. При этом тип этого кортежа определяется автоматически на основе типов данных, входящих в него. В связи с тем, что уровень доступа функции — private, его необходимо указать явно

private func someFunction() -> (SomeInternalClass, SomePrivateClass) {

// тело функции

}

Что касается перечислений, стоит обратить внимание на то, что каждый член перечисления получает тот же уровень доступа, что установлен для самого перечисления.

**Универсальные шаблоны**

Универсальные шаблоны (generic) являются одним из мощнейших инструментов Swift. На их основе написано большинство библиотек. Примером может служить тип данных Array (массив). Элементами массива могут выступать значения произвольных типов данных, и для этого разработчикам не требуется создавать отдельные типы массивов: Array<Int>, Array<String> и т.д. Для реализации коллекции использован универсальный шаблон, позволяющий при необходимости указать требования к типу данных.

Универсальные функции

Универсальные функции объявляются точно так же, как и стандартные, за одним исключением: после имени функции в угловых скобках указывается заполнитель имени типа, то есть литерал, который далее в функции будет указывать на тип данных переданного аргумента.

func swapTwoValues<T>( a: inout T, b: inout T) {

let temporaryA = a

a = b

b = temporaryA

}

var firstString = "one"

var secondString = "two"

swapTwoValues(a: &firstString, b: &secondString)

Используемый заполнитель называется параметром типа. Как только вы его определили, можете применять его для указания типа любого параметра или значения, включая возвращаемое функцией значение. При необходимости можно задать несколько параметров типа, вписав их в угловых скобках через запятую.

Универсальные типы

В дополнение к универсальным функциям универсальные шаблоны позволяют создать универсальные типы данных. К универсальным типам относятся, например, упомянутые ранее массивы и словари.

struct Stack<T> {

var items = [T]()

mutating func push(\_ item: T) {

items.append(item)

}

mutating func pop() -> T {

return items.removeLast()

}

}

Универсальная версия отличается от неуниверсальной только тем, что вместо указания конкретного типа данных задается заполнитель имени типа . Создавая новую коллекцию типа Stack, в угловых скобках необходимо указать тип данных, после чего можно использовать описанные методы для модификации хранилища

var stackOfStrings = Stack<String>()

stackOfStrings.push("uno")

stackOfStrings.push("dos")

let fromTheTop = stackOfStrings.pop()

Swift позволяет написать расширение для универсальных структур, классов и протоколов. Расширяя общий тип, не нужно указывать параметр типа. Тем не менее, можно использовать тип заполнителя в расширении.

extension Stack {// нет типа параметра

var topItem: T? {

return items.last

}

}

Ограничения типа

Иногда бывает полезно указать определенные ограничения, накладываемые на типы данных универсального шаблона. В качестве примера мы уже рассматривали тип данных Dictionary, где для ключа существует требование: тип данных должен соответствовать протоколу Hashable. Универсальные шаблоны позволяют накладывать определенные требования и ограничения на тип данных значения. Вы можете указать список типов, которым должен соответствовать тип значения. Если элементом этого списка является протокол (который также является типом данных), то проверяется соответствие типа значения данному протоколу; если типом является класс, структура или перечисления, то проверяется, соответствует ли тип значения данному типу. Для определения ограничений необходимо передать перечень имен типов через двоеточие после заполнителя имени типа. Реализуем функцию, производящую поиск элемента в массиве и возвращающую его индекс

func findIndex<T: Equatable>(array: [T], valueToFind: T) -> Int? {

for (index, value) in array.enumerated() {

if value == valueToFind {

return index

}

}

return nil

}

var doubArray = [3.14159, 0.1, 0.25]

let firstIndex = findIndex(array: doubArray, valueToFind: 0.1) // 1

let secondIndex = findIndex(array: doubArray, valueToFind: 31) // nil

Параметр типа записывается как <T: Equatable>. Это означает «любой тип, поддерживающий протокол Equatable». В результате поиск в переданном массиве выполняется без ошибок, поскольку тип данных Int поддерживает протокол Equatable, следовательно, значения данного типа могут быть приняты к обработке.

Связанные типы

При определении протокола бывает удобно использовать связанные типы, указывающие на некоторый, пока неизвестный, тип данных. Связанный тип позволяет задать заполнитель типа данных, который будет использоваться при заполнении протокола. Фактически тип данных не указывается до тех пор, пока протокол не будет принят каким-либо объектным типом. Связанные типы указываются с помощью ключевого слова associatedtype, за которым следует имя связанного типа. Определим протокол Container, использующий связанный тип ItemType

protocol Container {

associatedtype ItemType

mutating func append(item: ItemType)

var count: Int { get }

subscript(i: Int) -> ItemType { get }

}

Протокол Container (контейнер) может быть задействован в различных коллекциях, например в описанном ранее типе коллекции Stack. В этом случае тип данных, используемый в свойствах и методах протокола, заранее неизвестен. Для решения проблемы используется связанный тип ItemType, который определяется лишь при принятии протокола типом данных.

struct Stack<T>: Container {

typealias ItemType = T

var items = [T]()

var count: Int {

return items.count

}

init(){}

init(\_ elements: T...){

self.items = elements

}

subscript(i: Int) -> T {

return items[i]

}

mutating func push(item: T) {

items.append(item)

}

mutating func pop() -> T {

return items.removeLast()

}

mutating func append(item: T) {

items.append(item)

}

}

Так как тип Stack теперь поддерживает протокол Container, в нем появилось три новых элемента: свойство, метод и сабскрипт. Ключевое слово typealias указывает на то, какой тип данных является связанным в данном объектном типе.

ПРИМЕЧАНИЕ. Обратите внимание на то, что при описании протокола используется ключевое слово associatedtype, а при описании структуры—typealias.

Так как заполнитель имени использован в качестве типа аргумента item свойства append и возвращаемого значения сабскрипта, Swift может самостоятельно определить, что заполнитель T указывает на тип ItemType, соответствующий типу данных в протоколе Container. При этом указывать ключевое слово associatedtype не обязательно: если вы его удалите, то тип продолжит работать без ошибок.

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

**Задание 9.1**

1. *Создайте универсальную структуру IssueList<T> c обощенным массивом, методами add (item:T), remove()🡪T.*
2. *Напишите пример работы с IssueList<Bug>.*
3. *Напишите расширение для IssueList - вычисляемое свойство lastIssue: T?, которое возвращает последний элемент в массиве.*
4. *Определеите протокол IssueStorage со связанным (*associatedtype*) типом Item и содержащим свойство для измеения и доступа к массиву, методы добавления и удаления Item к массиву.*
5. *Продемонстрируте на любом классе поддержку протокола IssueStorage*
6. *Продемонстрируйте в коде понимание модификаторов доступа*

Вопросы:

1. Перечислите и дайте пояснения для каждого из уровней разграничения доступа?
2. В чем отличие open и public?
3. Какой уровень доступа устанавливается по умолчанию?
4. Зависит ли уровень доступа членов типа от уровня доступа типа? Если да, то как? Приведите пример.
5. Как определяется уровень доступа функции?
6. Приведите пример универсального типа (generic) и универсальной функции.
7. Можно ли для универсального типа сделать расширение? Приведите пример.
8. Как сделать ограничение на тип в универсальном шаблоне?