Лабораторная работа №3

Тема занятия: «Использование языка программирования Swift: структуры и классы, методы»

Цель: Выполнить разработку приложения с использованием языка программирования Swift: структуры и классы, методы.

Время выполнения: 8 часа.

3.1 СТРУКТУРЫ И КЛАССЫ

Классы и структуры являются универсальными и гибкими конструкциями, которые станут строительными блоками для кода вашей программы. Для добавления функциональности в классах и структурах можно объявить свойства и методы, применив тот же синтаксис, как и при объявлении констант, переменных и функций.

В отличие от других языков программирования, *Swift* не требует создавать отдельные файлы для интерфейсов и реализаций пользовательских классов и структур. В *Swift*, вы объявляете структуру или класс в одном файле, и внешний интерфейс автоматически становится доступным для использования в другом коде.

Экземпляр *класса* традиционно называют *объектом*. Тем не менее, классы и структуры в *Swift* гораздо ближе по функциональности, чем в других языках, и многое в этой главе описывает функциональность, которую можно применить к экземплярам *и* класса, *и* структуры. В связи с этим, употребляется более общий термин – *экземпляр*.

3.1.1 Сравнение классов и структур

Классы и структуры в Swift имеют много общего. И в классах и в структурах можно:

1. Объявлять свойства для хранения значений
2. Объявлять методы, чтобы обеспечить функциональность
3. Объявлять индексы, чтобы обеспечить доступ к их значениям, через синтаксис индексов
4. Объявлять инициализаторы, чтобы установить их первоначальное состояние
5. Они оба могут быть расширены, чтобы расширить их функционал за пределами стандартной реализации
6. Они оба могут соответствовать протоколам, для обеспечения стандартной функциональности определенного типа

Классы имеют дополнительные возможности, которых нет у структур:

1. Наследование позволяет одному классу наследовать характеристики другого
2. Приведение типов позволяет проверить и интерпретировать тип экземпляра класса в процессе выполнения
3. Деинициализаторы позволяют экземпляру класса освободить любые ресурсы, которые он использовал

Дополнительные возможности поддержки классов связаны с увеличением сложности. Лучше использовать структуры и перечисления, потому что их легче понимать. Также не забывайте про классы. На практике - большинство пользовательских типов данных, с которыми вы работаете - это структуры и перечисления.

3.1.2 Синтаксис объявления

Классы и структуры имеют схожий синтаксис объявления. Для объявления классов, используйте ключевое слово class, а для структур – ключевое слово struct. В обоих случаях необходимо поместить все определение полностью внутрь пары фигурных скобок:

```
class SomeClass {
    // здесь пишется определение класса
}
struct SomeStructure {
    // здесь пишется определение структуры
}
```

Что бы вы не создавали, новый класс или структуру, вы фактически создаете новый тип в Swift. Назначайте имена типов, используя UpperCamelCase(SomeClass или SomeStructure), чтобы соответствовать стандартам написания имен типов в Swift (например, String, Int и Bool). С другой стороны, всегда назначайте свойствам и методам имена в lowerCamelCase (например, frameRate и incrementCount), чтобы отличить их от имен типов.

Пример определения структуры и класса:

```
struct Resolution {
    var width = 0
    var height = 0
}
class VideoMode {
    var resolution = Resolution()
    var interlaced = false
    var frameRate = 0.0
    var name: String?
}
```

Пример выше объявляет новую структуру Resolution для описания разрешения монитора в пикселях. Эта структура имеет два свойства width, height. Хранимые свойства - или константы, или переменные, которые сгруппированы и сохранены в рамках класса или структуры. Этим свойствам выведен тип Int, так как мы им присвоили целочисленное значение 0.

В примере мы так же объявили и новый класс VideoMode, для описания видеорежима для отображения на видеодисплее. У класса есть четыре свойства в виде переменных. Первое - resolution, инициализировано с помощью экземпляра структуры Resolution, что выводит тип свойства как Resolution. Для остальных трех свойств новый экземпляр класса будет инициализирован с interlaced = false, frameRate = 0.0 и опциональным значением типа String с названием name. Это свойство name автоматически

будет иметь значение nil или «нет значения для name», потому что это опциональный тип.

3.1.3 Экземпляры класса и структуры

Объявление структуры Resolution и класса VideoMode только описывают как Resolution и VideoMode будут выглядеть. Сами по себе они не описывают специфическое разрешение или видеорежим. Для того чтобы это сделать нам нужно создать экземпляр структуры или класса.

Синтаксис для образования экземпляра класса или структуры очень схож:

```
let someResolution = Resolution()
let someVideoMode = VideoMode()
```

И классы и структуры используют синтаксис инициализатора для образования новых экземпляров. Самая простая форма синтаксиса инициализатора - использование имени типа и пустые круглые скобки сразу после него Resolution(), VideoMode(). Это создает новый экземпляр класса или структуры с любыми инициализированными свойствами с их значениями по умолчанию.

3.1.4 Доступ к свойствам

Вы можете получить доступ к свойствам экземпляра, используя точечный синтаксис. В точечном синтаксисе имя свойства пишется сразу после имени экземпляра, а между ними вписывается точка (.) без пробелов:

```
print("The width of someResolution is \(someResolution.width)")
// Выведет "The width of someResolution is 0"
```

В этом примере someResolution.width ссылается на свойство width экземпляра someResolution, у которого начальное значение равно 0.

Вы можете углубиться в подсвойства, например, свойство width свойства resolution класса VideoMode:

```
print("The width of someVideoMode is \(someVideoMode.resolution.width)")
// Выведет "The width of someVideoMode is 0"
```

Вы так же можете использовать точечный синтаксис для присваивания нового значения свойству:

```
someVideoMode.resolution.width = 1280
print("The width of someVideoMode is now \(someVideoMode.resolution.width)")
```

```
// Выведет "The width of someVideoMode is now 1280"
```

3.1.5 Поэлементные инициализаторы структурных типов

Все структуры имеют автоматически генерированный "поэлементный инициализатор", который вы можете использовать для инициализации свойств новых экземпляров структуры. Начальные значения для свойств нового экземпляра могут быть переданы поэлементному инициализатору по имени:

```
let vga = Resolution(width: 640, height: 480)
```

В отличие от структур, классы не получили поэлементного инициализатора исходных значений.

3.2 СТРУКТУРЫ И ПЕРЕЧИСЛЕНИЯ – ТИПЫ ЗНАЧЕНИЯ

Тип значения – это тип, значение которого копируется, когда оно присваивается константе или переменной, или когда передается функции.

Вообще вы уже достаточно активно использовали типы на протяжении предыдущих глав. Но факт в том, что все базовые типы Swift - типы значений и реализованы они как структуры.

Все структуры и перечисления – типы значений в Swift. Это значит, что любой экземпляр структуры и перечисления, который вы создаете, и любые типы значений, которые они имеют в качестве свойств, всегда копируются, когда он передается по вашему коду.

Коллекции, определенные стандартной библиотекой, такие как массивы, словари и строки, используют оптимизацию для снижения затрат на копирование. Вместо того, чтобы немедленно сделать копию, эти коллекции совместно используют память, в которой элементы хранятся между исходным экземпляром и любыми копиями. Если одна из копий коллекции модифицирована, элементы копируются непосредственно перед изменением.

Рассмотрим пример, который использует структуру Resolution из предыдущего примера:

```
let hd = Resolution(width: 1920, height: 1080)
var cinema = hd
```

Этот пример объявляет константу hd и присваивает экземпляр Resolution, инициализированный двумя значениями width и height.

В свою очередь, объявляем переменную cinema и присваиваем ей текущее значение hd. Так как Resolution - структура, делается копия существующего экземпляра, и эта новая копия присваивается cinema. Даже не смотря на то, что hd и cinema имеют одни и те же height, width, они являются абсолютно разными экземплярами.

Следующим шагом изменим значение свойства width у cinema, мы сделаем его чуть больше 2 тысяч, что является стандартным для цифровой кинопроекции (2048 пикселей ширины на 1080 пикселей высоты):

```
cinema.width = 2048
```

Если мы проверим свойство width у cinema, то мы увидим, что оно на самом деле изменилось на 2048:

```
print("cinema is now \(cinema.width) pixels wide")
// Выведет "cinema is now 2048 pixels wide"
```

Однако свойство width исходного hd экземпляра осталось 1920:

```
print("hd is still \(hd.width) pixels wide")
// Выведет "hd is still 1920 pixels wide"
```

Когда мы присвоили cinema текущее значение hd, то значения, которые хранились в hd были скопированы в новый экземпляр cinema. И в качестве результата мы имеем два совершенно отдельных экземпляра, которые содержат одинаковые числовые значения. Так как они являются раздельными экземплярами, то установив значение свойства width у cinema на 2048 никак не повлияет на значение width у hd, это показано на рсиунке 3.1:

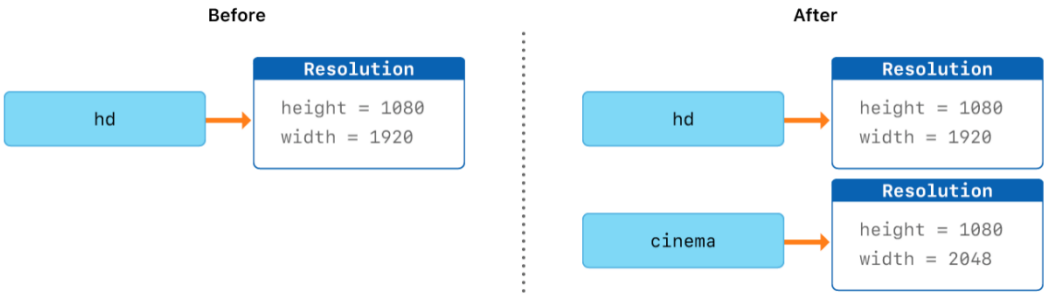

Рисунок 3.1 – Пример экземпляров структуры

То же поведение применимо к перечислениям:

```
enum CompassPoint {
    case north, south, east, west
    mutating func turnNorth() {
        self = .north
    }
}
var currentDirection = CompassPoint.west
let rememberedDirection = currentDirection
currentDirection.turnNorth()
print("Текущее направление - \(currentDirection)")
// Выведет "Текущее направление - north"
```

Когда мы присваиваем rememberedDirection значение currentDirection, мы фактически копируем это значение. Изменяя значение currentDirection, мы не меняем копию исходного значения, хранящейся в rememberedDirection.

3.3 КЛАССЫ – ССЫЛОЧНЫЙ ТИП

В отличие от типа значений, *ссылочный тип не копируется*, когда его присваивают переменной или константе, или когда его передают функции. Вместо копирования используется ссылка на существующий экземпляр.

Вот пример с использованием класса VideoMode, который был объявлен выше:

```
let tenEighty = VideoMode()
tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
tenEighty.frameRate = 25.0
```

В этом примере объявляем новую константу tenEighty и устанавливаем ссылку на новый экземпляр класса VideoMode. Значения видеорежима были присвоены копией со значениями 1920 на 1080. Мы ставим tenEighty.interlaced = true и даем имя “1080i”. Затем устанавливаем частоту кадров в секунду 25 .

Следующее, что мы сделаем, это tenEighty присвоим новой константе alsoTenEighty и изменим частоту кадров:

```
let alsoTenEighty = tenEighty
alsoTenEighty.frameRate = 30.0
```

Так как это классы ссылочного типа, то экземпляры tenEighty и alsoTenEighty ссылаются на один и тот же экземпляр VideoMode. Фактически получается, что у нас два разных имени для *одного* единственного экземпляра, как показано на рисунке 3.2:

Рисунок 3.2 – Пример экземпляров классов

Если мы проверим свойство frameRate у tenEighty, то мы увидим, что новая частота кадров 30.0, которая берется у экземпляра VideoMode:

```
print("The frameRate property of tenEighty is now \(tenEighty.frameRate)")
// Выведет "The frameRate property of tenEighty is now 30.0"
```

Этот пример показывает как сложно бывает отследить за ссылочными типами. Если tenEighty и alsoTenEighty находились бы в разных уголках вашей программы, то было бы сложно найти все места, где мы меняем режим воспроизведения видео. Где бы вы не использовали tenEighty, вам так же бы приходилось думать и о alsoTenEighty, и наоборот. В отличие от ссылочного типа, с типами значения дела обстоят значительно проще, так как весь код, который взаимодействует с одним и тем же значением, находится рядом, в вашем исходном файле.

Обратите внимание, что tenEighty и alsoTenEighty объявлены как константы, а не переменные. Однако вы все равно можете менять tenEighty.frameRate и alsoTenEighty.frameRate, потому что значения tenEighty и alsoTenEighty сами по себе не меняются, так как они не «содержат» значение экземпляра VideoMode, а напротив, они лишь ссылаются на него. Это свойство frameRate лежащего в основе VideoMode, которое меняется, а не значения константы ссылающейся на VideoMode.

3.3.1 Операторы тождественности

Так как классы являются ссылочными типами, то есть возможность сделать так, чтобы несколько констант и переменных ссылались на один единственный экземпляр класса. (Такое поведение не применимо к структурам и перечислениям, так как они копируют значение, когда присваиваются константам или переменным или передаются функциям.)

Иногда бывает полезно выяснить ссылаются ли две константы или переменные на один и тот же экземпляр класса. Для проверки этого в Swift есть два оператора тождественности:

– Идентичен (===)

– Не идентичен (!==)

Можно использовать эти операторы для проверки того, ссылаются ли две константы или переменные на один и тот же экземпляр:

```
if tenEighty === alsoTenEighty {
    print("tenEighty and alsoTenEighty refer to the same VideoMode instance.")
}
// Выведет "tenEighty and alsoTenEighty refer to the same VideoMode instance."
```

Обратите внимание, что «идентичность» (в виде трех знаков равенства, или ===) не имеет в виду «равенство» (в виде двух знаков равенства, или ==). Идентичность или тождественность значит, что две константы или переменные ссылаются на один и тот же экземпляр класса. Равенство значит,

что экземпляры равны или эквивалентны в значении в самом обычном понимании «равны».

Когда вы объявляете свой пользовательский класс или структуру, вы сами решаете, что означает "равенство" двух экземпляров. Процесс определения своей собственной реализации операторов "равенства" или "неравенства" описан в разделе Операторы эквивалентности.

3.3.2 Указатели

Если у вас есть опыт работы в C, C++ или Objective-C, то вы, может быть, знаете, что эти языки используют указатели для ссылки на адрес памяти. В Swift константы и переменные, которые ссылаются на экземпляр какого-либо ссылочного типа, аналогичны указателям C, но это не прямые указатели на адрес памяти, и они не требуют от вас написания звездочки(*) для индикации того, что вы создаете ссылку. Вместо этого такие ссылки объявляются как другие константы или переменные в Swift. Стандартная библиотека предоставляет типы указателей и буферов, которые вы можете использовать, если вам нужно напрямую взаимодействовать с указателями.

3.4 МЕТОДЫ

Методы – это функции, которые связаны с определенным типом. Классы, структуры и перечисления - все они могут определять методы экземпляра, которые включают в себя определенные задачи и функциональность для работы с экземпляром данного типа. Классы, структуры и перечисления так же могут определить методы типа, которые связаны с самим типом. Методы типа работают аналогично методам класса в Objective-C.

Дело в том, что структуры и перечисления могут определить методы в Swift, что является главным отличием от C или Objective-C. В Objective-C классы единственный тип, который может определять методы. В Swift вы можете выбирать, стоит ли вам определять класс, структуру или перечисление, и вы все равно, при любом раскладе, получаете возможность определения методов типа, который вы создадите.

3.5 МЕТОДЫ ЭКЗЕМПЛЯРА

Методы экземпляра являются функциями, которые принадлежат экземплярам конкретного класса, структуры или перечисления. Они обеспечивают функциональность этих экземпляров, либо давая возможность доступа и изменения свойств экземпляра, либо обеспечивая функциональность экземпляра в соответствии с его целью. Методы экземпляра имеют абсолютно одинаковый синтаксис как и функции, что описаны в разделе Функции.

Вы пишете метод экземпляра внутри фигурных скобок типа, которому он принадлежит. Метод экземпляра имеет неявный доступ ко всем остальным методам экземпляра и свойствам этого типа. Метод экземпляра может быть вызван только для конкретного экземпляра типа, которому он принадлежит. Его нельзя вызвать в изоляции, без существующего экземпляра.

Ниже пример, который определяет простой класс Counter, который может быть использован для счета количества повторений действия:

```
class Counter {
    var count = 0
    func increment() {
        count += 1
    }
    func increment(by amount: Int) {
        count += amount
    }
    func reset() {
        count = 0
    }
}
```

Класс Counter() определяет три метода экземпляра:

– increment увеличивает значение счетчика на 1

– increment(by: Int) увеличивает значение счетчика на определенное значение amount.

– reset() сбрасывает значение счетчика на 0.

Класс Counter так же определяет переменное свойство count, для отслеживания значения счетчика.

Вы можете вызвать методы экземпляра с тем же точечным синтаксисом:

```
let counter = Counter()
// начальное значение counter равно 0
counter.increment()
// теперь значение counter равно 1
counter.increment(by: 5)
// теперь значение counter равно 6
counter.reset()
// теперь значение counter равно 0
```

Параметры функций могут иметь и имя аргумента (для использования внутри функций), и ярлык аргумента (для использования при вызове функций). То же самое верно для имен параметров методов, потому как методы те же самые функции, но ассоциированные с определенным типом.

3.5.1 Свойство self

Каждый экземпляр типа имеет неявное свойство self, которое является абсолютным эквивалентом самому экземпляру. Вы используете свойство self для ссылки на текущий экземпляр, внутри методов этого экземпляра.

Метод increment может быть вызван так:

```
func increment() {
    self.count += 1
}
```

На практике вам не нужно писать self очень часто. Если вы не пишете self, то Swift полагает, что вы ссылаетесь на свойство или метод текущего экземпляра каждый раз, когда вы используете известное имя свойства или метода внутри метода. Это хорошо видно при использовании свойства count (а не self.count) внутри трех методов Counter.

Главное исключение из этого правила получается, когда имя параметра метода экземпляра совпадает с именем свойства экземпляра. В этой ситуации имя параметра имеет приоритет и появляется необходимость ссылаться на свойство в более подходящей форме. Вы используете свойство self для того, чтобы увидеть различие между именем параметра и именем свойства.

Здесь self разграничивает параметр метода x и свойство экземпляра, которое тоже x:

```
struct Point {
    var x = 0.0, y = 0.0
    func isToTheRightOf(x: Double) -> Bool {
        return self.x > x
    }
}
let somePoint = Point(x: 4.0, y: 5.0)
if somePoint.isToTheRightOf(x: 1.0) {
    print("Эта точка находится справа от линии, где x == 1.0")
}
// Выведет "Эта точка находится справа от линии, где x == 1.0"
```

Без префикса self, Swift будет думать, что в обоих случаях x будет как раз параметром метода, который так же называется x.

3.5.2 Изменение типов значений методами экземпляра

Структуры и перечисления являются *типами значений*. По умолчанию, свойства типов значений не могут быть изменены изнутри методов экземпляра.

Однако, если вам нужно изменить свойства вашей структуры или перечисления изнутри конкретного метода, то вы можете выбрать поведение как изменяющее для этого метода. После этого метод может изменить свои свойства изнутри метода, и все изменения будут сохранены в исходную структуру, когда выполнение метода закончится. Метод так же может присвоить совершенно новый экземпляр для свойства self, и этот новый экземпляр заменит существующий, после того как выполнение метода закончится.

Вы можете все это осуществить, если поставите ключевое слово mutating перед словом func для определения метода:

```
struct Point {
   var x = 0.0, y = 0.0
   mutating func moveBy(x deltaX: Double, y deltaY: Double)
{
      x += deltaX
      y += deltaY
   }
}
var somePoint = Point(x: 1.0, y: 1.0)
somePoint.moveBy(x: 2.0, y: 3.0)
print("Сейчас эта точка на (\(somePoint.x),
\(somePoint.y))")
// Выведет "Сейчас эта точка на (3.0, 4.0)"
```

Структура Point определяет метод moveBy(x:y:), который передвигает точку типа Point на определенное количество значений. Вместо того, чтобы вернуть новую точку, этот метод фактически изменяет координаты точки, которая его вызвала. Ключевое слово mutating добавлено к определению метода, для того, чтобы изменить значения свойств.

Обратите внимание, что вы не можете вызвать изменяющий (mutating) метод для константных типов структуры, потому как ее свойства не могут быть изменены, даже если свойства являются переменными.

```
let fixedPoint = Point(x: 3.0, y: 3.0)
fixedPoint.moveBy(x: 2.0, y: 3.0)
// это вызовет сообщение об ошибке
```

3.5.3 Присваивание значения для self внутри изменяющего метода

Изменяющие методы могут присваивать полностью новый экземпляр неявному свойству self. Пример Point, приведенный выше, мог бы быть записан в такой форме:

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveBy(x deltaX: Double, y deltaY: Double)
{
```

```
            self = Point(x: x + deltaX, y: y + deltaY)
        }
    }
```

Такая версия изменяющего метода moveBy(x:y:) создает абсолютно новую структуру, чьим значениям x, y присвоены значения конечной точки. Конечный результат вызова этой альтернативной версии метода будет абсолютно таким же как и в ранней версии.

Изменяющие методы для перечислений могут установить отдельный член перечисления как неявный параметр self:

```
enum TriStateSwitch {
    case off, low, high
    mutating func next() {
        switch self {
        case .off:
            self = .low
        case .low:
            self = .high
        case .high:
            self = .off
        }
    }
}
var ovenLight = TriStateSwitch.low
ovenLight.next()
// ovenLight равен .high
ovenLight.next()
// ovenLight равен .off
```

В этом примере мы рассматриваем перечисление с тремя положениями переключателя. Переключатель проходит последовательно три положения (off, low, high), каждый раз меняя положение, как вызывается метод next().

3.6 МЕТОДЫ ТИПА

Методы экземпляра, которые описаны выше, являются методами, которые вызываются экземпляром конкретного типа. Вы так же можете определить методы, которые вызываются самим типом. Такие методы зовутся *методами типа*. Индикатор такого метода – ключевое слово static, которое ставится до ключевого слова метода func. Классы так же могут использовать ключевое слово class, чтобы разрешать подклассам переопределение инструкций суперкласса этого метода.

В Objective-C определять методы типов можно только для классов. В Swift вы можете создавать методы типа не только для классов, но и для структур и перечислений. Метод каждого типа ограничен самим типом, который его поддерживает.

Такие методы так же используют точечный синтаксис, как и методы экземпляра. Однако эти методы вы вызываете самим типом, а не экземпляром этого типа. Вот как вы можете вызвать метод самим классом SomeClass:

```
class SomeClass {
    class func someTypeMethod(){
        //здесь идет реализация метода
    }
}

SomeClass.someTypeMethod()
```

Внутри тела метода типа неявное свойство self ссылается на сам тип, а не на экземпляр этого типа. Это значит, что вы можете использовать self для того, чтобы устранить неоднозначность между свойствами типа и параметрами метода типа, точно так же как вы делали для свойств экземпляра и параметров метода экземпляра.

Если обобщить, то любое имя метода и свойства, которое вы используете в теле метода типа, будет ссылаться на другие методы и свойства на уровне типа. Метод типа может вызвать другой метод типа с иным именем метода, без использования какого-либо префикса имени типа. Аналогично, методы типа в структурах и перечислениях могут получить доступ к свойствам типа, используя имя этого свойства, без написания префикса имени типа.

Пример ниже определяет структуру с именем LevelTracker, которая отслеживает прогресс игрока на разных уровнях игры. Это одиночная игра, но может хранить информацию для нескольких игроков на одном устройстве.

Все уровни игры (кроме первого уровня) заблокированы, когда играют в первый раз. Каждый раз, заканчивая уровень, этот уровень открывается и у остальных игроков на устройстве. Структура LevelTracker использует свойства и методы типа для отслеживания уровней, которые были разблокированы. Так же она отслеживает текущий уровень каждого игрока.

```
struct LevelTracker {
    static var highestUnlockedLevel = 1
    var currentLevel = 1

    static func unlock(_ level: Int) {
        if level > highestUnlockedLevel {
highestUnlockedLevel = level }
    }

    static func isUnlocked(_ level: Int) -> Bool {
        return level <= highestUnlockedLevel
    }

    @discardableResult mutating func advance(to level: Int)
-> Bool {
        if LevelTracker.isUnlocked(level) {
            currentLevel = level
```

```
            return true
        } else {
            return false
        }
    }
}
```

Структура LevelTracker следит за самым последним уровнем, который разблокировал игрок. Это значение лежит в свойстве типа highestUnlockedLevel.

LevelTracker так же определяет две функции для работы со свойством highestUnlockedLevel. Первая функция типа unlock(_:), которая обновляет значение highestUnlockedLevel, каждый раз когда открывается новый уровень. Вторая функция типа isUnlocked(_:), которая возвращает true, если конкретный уровень уже разблокирован. (Обратите внимание, что методы типа могут получить доступ к highestUnlockedLevel без написания LevelTracker.highestUnlockedLevel.)

В дополнение к его свойствам типа и методам типа, структура LevelTracker так же отслеживает и текущий прогресс игрока в игре. Она использует свойство экземпляра currentLevel для отслеживания уровня, на котором игрок играет.

Для помощи в управлении свойством currentLevel, структура LevelTracker определяет метод экземпляра advance(to:). До того как обновить currentLevel, этот метод проверяет доступен ли запрашиваемый новый уровень. Метод advance(to:) возвращает логическое значение, указывающее, удалось ли ему поставить currentLevel. Не обязательно должно быть ошибкой игнорирование результата работы функции advance(to:), поэтому этот метод имеет маркировку @discardableResult.

Структура LevelTracker используется классом Player, который описан ниже, для отслеживания и обновления прогресса конкретного игрока:

```
class Player {
    var tracker = LevelTracker()
    let playerName: String
    func complete(level: Int) {
        LevelTracker.unlock(level + 1)
        tracker.advance(to: level + 1)
    }
    init(name: String) {
        playerName = name
    }
}
```

Класс Player создает новый экземпляр LevelTracker для отслеживания прогресса игрока. Так же он определяет и использует метод complete(level:), который вызывается каждый раз, как игрок заканчивает уровень. Этот метод открывает следующий уровень для всех игроков. (Логическое значение

advance(to:) игнорируется, так как уровень открывается функцией LevelTracker.unlock(_:) на предыдущей строке.)

Вы можете создать экземпляр класса Player для нового игрока и увидеть, что будет, когда игрок закончит первый уровень:

```
var player = Player(name: "Argyrios")
player.complete(level: 1)
print("Самый последний доступный уровень сейчас равен \(LevelTracker.highestUnlockedLevel)")
//Выведет "Самый последний доступный уровень сейчас равен 2"
```

Если вы создадите второго игрока, и попробуете им начать прохождение уровня, который не был разблокирован ни одним игроком в игре, то вы увидите, что эта попытка будет неудачной:

```
player = Player(name: "Beto")
if player.tracker.advance(to: 6) {
    print("Игрок на уровне 6")
} else {
    print("Уровень 6 еще не разблокирован")
}
// Выведет "Уровень 6 еще не разблокирован"
```

Вопросы к лабораторной работе № 3

1. Что такое класс?
2. Что такое структура?
3. Чем класс отличается от структуры?
4. Что такое объект?
5. Типы значений. Ссылочные типы. Чем отличаются друг от друга?
6. Для чего используются указатели?
7. Ключевое слово self.

Задание к лабораторной работе № 3

Задание к лабораторной работе:

– согласно варианта задания написать программу (Вариант = номер в списке подгруппы % 15 + 1);

– разбить функционал приложения на несколько пакетов придерживаясь логики.

– сделать валидацию всех вводимых значений.

Варианты

1 Строки сравниваются на основании значений символов. Т.е. если мы захотим выяснить, что больше: «Apple» или «Яблоко», – то «Яблоко» окажется бОльшим. Такое положение дел не устроило Анну. Она считает, что строки нужно сравнивать по количеству входящих в них символов. Для этого девушка создала класс RealString и реализовала озвученный инструментарий. Сравнивать между собой можно как объекты класса, так и обычные строки с экземплярами класса RealString.

2 Николай – оригинальный человек. Он решил создать класс Nikola, принимающий при инициализации 2 параметра: имя и возраст. Но на этом он не успокоился. Не важно, какое имя передаст пользователь при создании экземпляра, оно всегда будет содержать «Николая». В частности – если пользователя на самом деле зовут Николаем, то с именем ничего не произойдет, а если его зовут, например, Максим, то оно преобразуется в “Я не Максим, а Николай”.

3 Создайте класс ПЕРСОНА с методами, позволяющими вывести на экран информацию о персоне, а также определить ее возраст (в текущем году). Создайте дочерние классы: АБИТУРИЕНТ (фамилия, дата рождения, факультет), СТУДЕНТ (фамилия, дата рождения, факультет, курс), ПРЕПОДАВАТЕЛЬ (фамилия, дата рождения, факультет, должность, стаж), со своими методами вывода информации на экран и определения возраста. Создайте список из п персон, выведите полную информацию из базы на экран, а также организуйте поиск персон, чей возраст попадает в заданный диапазон.

4 Напишите программу с классом Student, в котором есть три атрибута:name, groupNumber и age. По умолчанию name = Ivan, age = 18, groupNumber = 10A. Необходимо создать пять методов: getName, getAge, getGroupNumber, setNameAge, setGroupNumber. Метод getName нужен для получения данных об имени конкретного студента, метод getAge нужен для получения данных о возрасте конкретного студента, vетод setGroupNumberнужен для получения данных о номере группы конкретного студента. Метод SetNameAge позволяет изменить данные атрибутов, установленных по умолчанию, метод setGroupNumber позволяет изменить номер группы, установленный по умолчанию. В программе необходимо создать пять экземпляров класса Student, установить им разные имена,

возраст и номер группы.

5 Напишите программу с классом Math. Создайте два атрибута – a и b. Напишите методы addition – сложение, multiplication – умножение, division – деление, subtraction – вычитание. При передаче в методы параметров a и b с ними нужно производить соответствующие действия и печатать ответ.

6 Напишите программу с классом Car. Создайте конструктор класса Car. Создайте атрибуты класса Car – color (цвет), type (тип), year (год). Напишите пять методов. Первый – запуск автомобиля, при его вызове выводится сообщение «Автомобиль заведен». Второй – отключение автомобиля – выводит сообщение «Автомобиль заглушен». Третий – присвоение автомобилю года выпуска. Четвертый метод – присвоение автомобилю типа. Пятый – присвоение автомобилю цвета.

7 Магические квадраты издавна интриговали воображение людей: дата изготовления древнейшей сохранившейся таблицы относится к 2200 г. До н.э. Магический квадрат – это квадратная таблица размера n x n, составленная из всех чисел 1, 2, 3 … n2 таким образом, что суммы по каждому столбцу, каждой строке и каждой диагонали равны между собой. Напишите программу (используя классы), которая определяет, можно ли считать матрицу магическим квадратом.

8 Реализовать задачу, используя классы. На шахматной доске 8 x 8 стоит ферзь. Отметьте положение ферзя на доске и все клетки, которые бьет ферзь. Клетку, где стоит ферзь, отметьте буквой Q, клетки, которые бьет ферзь, отметьте звездочками *, остальные клетки заполните точками. Шахматный ферзь может ходить по вертикали, горизонтали и по диагоналям. Входные данные: Координаты ферзя на шахматной доске в формате номер столбца (буква от a до h, слева направо) и номер строки (цифра от 1 до 8, снизу вверх). Пример ввода: c4. Выходные данные: программа выводит стилизованное изображение шахматной доски со схемой возможных передвижений ферзя.

9 Экземляр класса задается тройкой координат в трехмерном пространстве (x,y,z). Обязательно должны быть реализованы методы: приведение вектора к строке с выводом координат, сложение векторов, вычитание векторов, скалярное произведение, умножение и деление на скаляр, векторное произведение, вычисление длинны вектора .

10 Класс содержит поле для задания количества элементов и поле для хранения элементов массива. Методы: конструктор без параметров, конструктор с параметрами, конструктор копирования, ввод и вывод данных, поиск максимального и минимального элементов, сортировка массива, поиск суммы элементов, метод умножения элементов массива на число.

11 При инициализации класс принимает целое нечетное число – сторону квадрата, в который вписана снежинка. Методы: thaw() – таять, при этом на каждом шаге пропадают крайние звездочки со всех сторон; параметр показывает, сколько шагов прошло; freeze(n) – замораживаться, при этом

сторона квадрата, в который вписана снежинка, увеличивается на 2 * n, одновременно добавляются звездочки в нужных местах, чтобы правило соблюдалось; thicken() – утолщаться, ко всем линиям звездочек с двух сторон добавляются параллельные (если перед этим снежинка таяла, то теперь звездочки восстанавливаются); show() – показывать (рисуется снежинка в виде квадратной матрицы со звездочками и дефисами в пустых местах).

12 Экземпляр класса инициализируется с аргументом name – именем котенка. Класс реализует методы: toAnswer() – ответить: котенок через один раз отвечает да или нет, начинает с да. Метод возвращает «moore-moore», если да, «meow-meow», если нет. Одновременно увеличивается количество соответствующих ответов; numberYes() – количество ответов да; numberNo() – количество ответов нет.

13 Экземпляр класса при инициализации принимает аргументы: имя, должность и стаж работы сотрудника, метод printInfo() выводит информацию о сотруднике в формате: «Имя: Василий Должность: Системный администратор Стаж: 3 года» При выводе стажа нужно учитывать, что «года» должно заменяться на «лет» или «год» в зависимости от числа.

14 Класс «Товар» содержит следующие закрытые поля: название товара, название магазина в котором продается товар стоимость товара в рублях Класс «Склад» содержит закрытый массив товаров. Обеспечить следующие возможности: вывод информации о товаре по номеру с помощью индекса; вывод информации о товаре, название которого введено с клавиатуры; сортировку товаров по названию магазина, по наименованию и цене; операцию сложения товаров, выполняющую сложение их цен.

15 Экземпляр класса имеет координаты своего положения и угол, описывающий направление движения. Он может быть изначально поставлен в любую точку с любым направлением (конструктор), может проехать в выбранном направлении определённое расстояние и может повернуть, то есть изменить текущее направление на любое другое. Реализуйте класс автомобиля, а также класс, который будет описывать автобус. Кроме того, что имеется у автомобиля, у автобуса должны быть поля, содержащие число пассажиров и количество полученных денег, изначально равные нулю. Также должны быть методы «войти» и «выйти», изменяющие число пассажиров. Метод «move» должен увеличивать количество денег в соответствии с количеством пассажиров и пройденным расстоянием.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Изучить теоретическую часть лабораторной работы.

2. Реализовать индивидуальное задание по вариантам, сделать скриншоты работающих программ. Написать комментарии.

3. Написать отчет, содержащий:

1. Титульный лист, на котором указывается:

а) полное наименование министерства образование и название

учебного заведения;

б) название дисциплины;

в) номер практического занятия;

г) фамилия преподавателя, ведущего занятие;

д) фамилия, имя и номер группы студента;

е) год выполнения лабораторной работы.

2. Индивидуальное задание с кодом, комментариями и скриншотами работающих программ.

3. Вывод по проделанной работе.