Лабораторная работа 3. Классы

Класс – это языковая конструкция, которая объединяет в себе функции и переменные, относящиеся к какой-либо сущности. Например, класс `Rectangle` (прямоугольник) может содержать переменные с шириной и высотой прямоугольника, а также метод для вычисления площади.

Функции, которые входят в класс, называют **методами**. Поэтому обычно говорят «метод, вычисляющий площадь прямоугольника», а не «функция, вычисляющая площадь прямоугольника». Хотя это одно и то же, и, в принципе, оба варианта не будут ошибкой. То же самое относится и к переменным – они называются **полями**: «поле содержит ширину прямоугольника».

Из классов воздаются **объекты** – конкретные экземпляры класса. В этом смысле класс можно сравнить с чертежом или схемой, по которой будет произведено устройство, а объекты – это сами устройства, которые штампуются на конвейере на основе схемы. Объектов из одного класса можно произвести много, и каждый из них будет независим от другого такого объекта.

Чтобы описать класс в Kotlin, используется следующая конструкция:

```
class Figure {
}
```

Это уже полноценный класс, хотя в нём пока ещё ничего нет. Для такого пустого класса можно даже опустить фигурные скобки:

```
class Figure
```

Создать объект класса `Figure` можно следующим образом:

```
val f1 = Figure()
val f2 = Figure()
```

Теперь есть два экземпляра класса `Figure`, каждый со своим именем.

В класс можно добавить поля и методы:

```
class Rectangle {

    var width: Int = 0

    var height: Int = 0

    fun area(): Int {
        return width * height
    }
}
```

Так как поля и методы находятся в классе, к ним нельзя обратиться просто так, сначала нужно создать объект:

```
val r = Rectangle()
```

После этого можно обращаться к полям и методам объекта, указывая имя объекта, а после точки – поле или метод:

```
r.width = 20
r.height = 15
```

```
println(r.area()) // 300
```

Свойства

Так как поле – это обычная переменная, записать в ней можно любое значение, подходящее по типу. Поэтому иногда желательно не допускать записи значений, которые выходят за пределы допустимого диапазона. В таких случаях можно использовать **свойства** – это поля, для которых при записи или чтении значения может вызываться программный код.

Например, в классе может быть свойство, хранящее возраст человека. Возраст – это переменная целого типа, которая не должна быть отрицательной.

```
class Person {

    var age: Int = 0
        get() { return field }
        set(value) {
            if (value >= 0)
                field = value
        }

}
```

Здесь переменная age – это свойство, у которого есть геттер и сеттер. **Геттер** (от англ. *get*) – это код, который вызывается для получения значения. **Сеттер** (от англ. *set*) – код, который вызывается при присвоении значения.

Для каждого свойства есть автоматическая переменная `field`, которая хранит значение этого свойства. Например, для свойства `age` переменная `field` будет хранить возраст, для свойства `name` – имя, и т. д. Эта автоматическая переменная доступна только внутри геттеров и сеттеров, а снаружи к свойству можно обращаться просто по его имени.

В приведенном выше классе геттер просто возвращает значение поля, а сеттер – сначала проверяет является ли присваиваемое значение нулевым или положительным, и только в этом случае записывает значение в свойство.

Свойство может быть доступно только для чтения – в этом случае ему нельзя будет присвоить никакое значение. Как правило, это **вычисляемые свойства** – то есть свойства, значения которых динамически формируются на основании других данных. Например, если в классе хранятся имя, фамилия и отчество, то можно добавить свойство `fullname`, которое будет соединять их вместе:

```
class Person {
    var firstname: String = ""  // Имя
    var lastname: String = ""   // Фамилия
    var patronymic: String = "" // Отчество

    val fullname: String
        get() {
            return "$lastname $firstname $patronymic"
        }
}
```

У свойства `fullname` нет сеттера, только геттер. Поэтому оно объявляется как неизменяемая переменная (`val`). Используется свойство как обычная переменная:

```
val p = Person()
p.firstname = "Иван"
```

```
p.lastname = "Сидоров"
p.patronymic = "Петрович"
println(p.fullname) // Сидоров Иван Петрович
```

Модификаторы видимости

Бывает удобно хранить данные в переменных, которые не видны извне класса. Для этого используются **модификаторы видимости** – ключевые слова, которые разрешают или ограничивают доступ к полям и методам класса. В Kotlin предусмотрены четыре типа видимости:

- `public` – поле или метод видимы из любой точки кода
- `private` – поле или метод видимы только внутри класса
- `protected` – поле или метод видимы только внутри класса, или из класса-наследника
- `internal` – поле или метод видимы только внутри модуля (всех файлов, которые компилируются вместе, как правило это весь проект).

Если модификатор видимости у поля или метода не указан, то подразумевается публичный доступ (`public`).

Например, можно спрятать какие-то поля и методы, которые используются во внутренних вычислениях, но не должны быть видны снаружи класса, следующим образом:

```
class Person {
    private var isHealthOk = true
    ...
}

val p = Person()
println(p.isHealthOk) // Ошибка, поле не доступно
```

Конструкторы

При создании объекта можно выполнять какие-либо действия: например, инициализировать переменные, выполнять начальные расчёты и т. д. Для этого используются **конструкторы** – особые методы класса, вызываемые при создании объекта.

Код основного конструктора размещается в блоке инициализации:

```
class Person(Name: String) {
      ...
      init {
            name = Name
      }
}
```

Когда создаётся новый экземпляр класса – объект, автоматически вызывается блок инициализации, в котором можно произвести начальные настройки и выполнить необходимые действия. При этом блок инициализации имеет доступ и к тем переменным, которые передаются при создании объекта, они указываются в круглых скобках после имени класса.

Создаётся такой объект как обычно:

```
var ivan = Person("Иван")
```

Иногда бывает нужно иметь несколько конструкторов, отличающихся набором параметров. Например, один конструктор класса `Person` принимает только имя, а возраст при этом задаётся по умолчанию, а другой – принимает и имя, и возраст:

```
class Person(Name: String) {
```

```
      init {
            name = Name
      }

      constructor(Name: String, Age: Int) : this(Name) {
        age = Age
    }
}
```

В этом случае вторичный конструктор (или несколько конструкторов) задаётся ключевым словом `constructor`. После списка параметров пишется ключевое слово `this` – это обязательный вызов базового конструктора, который задан в блоке `init`. Сначала будет вызван блок инициализации, а затем уже выполнен вторичный конструктор. После этого можно использовать тот или иной конструктор по необходимости:

```
var ivan = Person("Иван")
var ivan = Person("Петр", 25)
```

Наследование классов и переопределение функций

Часто у нескольких классов бывают одинаковые поля и функции. В таких случаях бывает удобно создать базовый класс, который будет включать такие поля, а у дочерних классов (или, как их ещё называют, классов-наследников) добавить уникальные для него поля и функции. Например, класс `Person` может содержать ФИО человека, а дочерние классы `Student` и `Teacher` – номер группы и должность соответственно:

```
open class Person {
      var firstName: String = ""
      var middleName: String = ""
      var lastName: String = ""
}

class Student : Person() {
      var group: String = ""
}

class Teacher : Person() {
      var job: String = ""
}
```

Базовый класс обязательно должен описываться со словом open – если его нет, то наследовать от такого класса не получится. В дочерних классах после имени класса приводится имя класса-родителя. Дочерний класс наследует все свойства и функции родительского класса, и может добавлять свои.

Аналогичным образом могут переопределяться и функции, если они описаны с ключевым словом open:

```
open class Person {
      open fun greet() {
            print "Привет!"
      }
}

class Teacher : Person() {
```

```
    override fun greet() {
        print "Добрый день!"
    }
}
```

В этом примере у базового класса Person есть функция greet, которая показывает приветствие. А дочерний класс Teacher переопределяет эту функцию, делая приветствие более формальным.

Перечисления

Существует специальная разновидность классов, называемая перечислением. Такие классы, обозначаемые ключевым словом enum, содержат перечень констант:

```
enum class Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
    FRIDAY, SATURDAY, SUNDAY
}
```

Можно использовать такие классы-перечисления, чтобы задавать значения в коде. Например, вместо того чтобы писать дни недели числами (1 – понедельник, 2 – вторник, и т. д.) лучше использовать названия дней недели:

```
val day: Day = Day.FRIDAY
```

Это позволит избежать трудноуловимых ошибок, происходящих из-за случайно перепутанного числа. Словесные названия обычно более понятны и легче проверяются.

Разработать систему классов (родительский и несколько дочерних), содержащих поля, свойства, функции и несколько конструкторов. Родительский класс должен определять общие для всех классов поля и функции. Дочерние классы должны переопределять какие-либо функции базового класса. Продемонстрировать в программе использование этих классов.
Задание (по вариантам):

1. Создайте структуру с именем student, содержащую поля: фамилия и инициалы, номер группы, успеваемость (массив из пяти элементов). Создать массив из десяти элементов такого типа, упорядочить записи по возрастанию среднего балла. Добавить возможность вывода фамилий и номеров групп студентов, имеющих оценки, равные только 4 или 5.

2. Создайте структуру с именем train, содержащую поля: название пункта назначения, номер поезда, время отправления. Ввести данные в массив из пяти элементов типа train, упорядочить элементы по номерам поездов. Добавить возможность вывода информации о поезде, номер которого введен пользователем. Добавить возможность сортировки массив по пункту назначения, причем поезда с одинаковыми пунктами назначения должны быть упорядочены по времени отправления.

3. Создать класс с двумя переменными. Добавить функцию вывода на экран и функцию изменения этих переменных. Добавить функцию, которая находит сумму значений этих переменных, и функцию которая находит наибольшее значение из этих двух переменных.

4. Описать класс, реализующий десятичный счетчик, который может увеличивать или уменьшать свое значение на единицу в заданном диапазоне. Предусмотреть инициализацию счетчика значениями по умолчанию и произвольными значениями. Счетчик имеет два метода: увеличения и уменьшения, — и свойство, позволяющее получить его текущее состояние. Написать программу, демонстрирующую все возможности класса.

5. Создать класс с двумя переменными. Добавить конструктор с входными параметрами. Добавить конструктор, инициализирующий члены класса по умолчанию. Добавить деструктор, выводящий на экран сообщение об удалении объекта.

6. Создать класс, содержащий динамический массив и количество элементов в нем. Добавить конструктор, который выделяет память под заданное количество элементов, и деструктор. Добавить методы, позволяющие заполнять массив случайными числами, переставлять в данном массиве элементы в случайном порядке, находить количество различных элементов в массиве, выводить массив на экран.

7. Составить описание класса многочленов от одной переменной, задаваемых сте пенью многочлена и массивом коэффициентов. Предусмотреть методы для вы числения значения многочлена для заданного аргумента, операции сложения, вычитания и умножения многочленов с получением нового объекта-многочлена, вывод на экран описания многочлена.