1.4. Объектно-ориентированное программирование

Сайт: Samsung Innovation Campus
Курс: Мобильная разработка на Kotlin
Книга: 1.4. Объектно-ориентированное программирование

Напечатано:: Murad Rezvan
Дата: понедельник, 3 июня 2024, 17:41

Оглавление

1.4.1. Основные принципы ООП

Основные принципы ООП.

Постоянный рост сложности программ в 1960-х привёл к необходимости в новом подходе, который бы позволил упростить разработку. В процессе разработки программистам стало слишком сложно удерживать в памяти детали реализации и те неявные связи, при помощи которых одни компоненты программы влияют на другие.

Новая технология получила название — объектно-ориентированное программирование (ООП). Это основная методология программирования 1990-х годов и начала XXI века. Она представляет собой продукт более 35 лет практики и опыта, которые восходят к использованию языка Simula 67. Идея, которую предложили разработчики этого языка оказалась судьбоносной в истории развития языков программирования. Позже на этих идеях были построены самые известные объектно-ориентированные языки, такие как Java, C++, а в последние годы и Kotlin,

В реальном мире нас повсюду окружают объекты. Мы учимся в аудиториях, перемещаемся на транспорте, работаем на компьютерах, звоним по телефонам и т. д. Все это — объекты реального мира. В ООП важно понимать, что описание задачи нужно вести в терминах объектов.

Центральной идеей ООП является реализация понятия «абстракция данных». Смысл абстракции заключается в том, что сущность можно рассматривать, а также производить определенные действия над ней, как над единым целым, не вдаваясь в детали внутреннего построения. Абстракция означает разработку классов исходя из их интерфейсов и функциональности, не принимая во внимание реализацию деталей.

Например, с помощью типа Int мы можем описать одно число целого типа, но при этом мы не можем сказать, что это будет за число: возраст человека, номер дома, страница книги. В нашей повседневной жизни существует огромный разрыв между таким примитивным понятием, как число, и сложными сущностями: «Юнит в компьютерной игре», «Персональная страничка человека в социальной сети», «Счет в банке» и т. д. На практике ООП сводится к написанию некоторого количества классов (то есть типов) и последующему их использованию.

С точки зрения ООП каждую такую сущность можно описать как совокупность:

- полей класса — переменных для хранения данных, описывающих класс. Это те свойства, параметры, характеристики, которые описывают состояние сущности;
- методов класса — функций для работы с полями класса. Это те действия, которые можно производить с этой сущностью.

Например, класс «Юнит в компьютерной игре» можно охарактеризовать как совокупность полей: «Имя», «Здоровье», «Броня», «Манна», «Количество патронов» и т. д. и методов: «Атаковать», «Защищаться», «Использовать манну» и т. д.

В итоге, если описать такой класс, мы получаем возможность работать со всем этим набором, как с единым целым, а не отдельными примитивными типами.

Так же, как мы объявляем переменную примитивного типа, мы можем создать переменную класса — объект. Объект — это экземпляр класса. Можно привести аналогию: класс — это форма для выпечки печений, а объект — это сами печенья.

В основе ООП лежат три принципа:

- инкапсуляция (от слова encapsulation или incapsulation) — объединение данных и процедур для работы с этими данными в единое целое;
- наследование (от слова inheritance) — средство получения новых классов из существующих;
- полиморфизм (от слова polymorphism) — создание общего интерфейса для группы близких по смыслу действий.

Инкапсуляция (encapsulation) — это механизм, который объединяет данные и код, манипулирующий этими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования. Инкапсуляция означает, что методы и поля класса рассматриваются в качестве единого целого. В связи с чем инкапсуляция требует соблюдения следующих принципов:

- все характеристики, которые описывают состояние объекта, должны храниться в его полях;
- все действия, которые можно осуществлять с объектом, должны описываться его методами;
- нельзя извне менять поля объекта.

Обратиться к полям объекта можно только при помощи методов. Из-за чего инкапсуляцию часто связывают с понятием «сокрытие данных». Такой подход позволяет, с одной стороны, обеспечить правильное функционирование внутренней структуры объекта. С другой стороны — при необходимости изменить режим внутренней работы объекта, не меняя способы его внешнего использования. Достаточно изменить внутреннюю структуру.

Например, объект реального мира — наручные часы. Для того чтобы ими пользоваться, нужно уметь их заводить и знать, какое время они показывают. Пользователю не нужно знать, как работает часовой механизм.

Если пользователю позволить менять местами шестеренки, то часы он, скорее всего, сломает.

С другой стороны, если часовщик изменит принцип внутренней работы наручных часов, пользователь этого не почувствует — в его распоряжении все также останется циферблат со стрелками, которые показывают время.

Наручные часы в этом примере (как и инкапсулированный объект) — это «черный ящик» для пользователя. Таким образом, ООП дает возможность пользователю работать с объектами, не задумываясь об их внутреннем устройстве.

Наследование — механизм языка, позволяющий описать новый класс на основе уже существующего (родительского, базового) класса. Класс-потомок может добавить собственные методы и свойства, а также пользоваться родительскими методами и свойствами. Позволяет строить иерархии классов.

Некоторые объекты являются частным случаем более общей категории объектов. Например, вы разрабатываете компьютерную игру. В игре должны быть предусмотрены различные персонажи: люди, роботы, волшебники и т. д. С одной стороны, каждый из объектов обладает отдельными характеристиками. С другой стороны, они имеют общие свойства (например, поля «Имя», «Здоровье», «Броня»). Хотелось бы иметь возможность рассматривать их как объект общей категории «Юнит», так как для всех этих объектов применимы общие действия (например, «Атаковать» и «Защищаться»), но при этом уметь хранить для каждого в отдельности его особенные свойства и применять к ним присущие каждому специфические методы.

Полиморфизм — это реализация одинакового по смыслу действия различным способом в зависимости от типа объекта.

Например, в предыдущем разделе мы рассмотрели метод «Атаковать», общий для всех юнитов компьютерной игры.

Несмотря на то что действие общее, каждый наследник реализует его по-своему. Например, человек использует для атаки меч, робот использует лазер, а волшебник — магическое заклинание. В этом заключается идея полиморфного поведения наследников по отношению к родительскому объекту.

При вызове метода «Атаковать» родительского класса для объектов разного типа компилятор сам понимает, какой метод нужно использовать.

1.4.2. Объявление классов в Kotlin

Классы в Kotlin объявляются с помощью использования ключевого слова class:

```
class Invoice {
}
```

Open in Playground → Target: JVM Running on v.2.0.0

Объявление класса состоит из имени класса, заголовка (указания типов его параметров, основного конструктора и т.п) и тела класса, заключённого в фигурные скобки. И заголовок, и тело класса являются необязательными составляющими: если у класса нет тела, фигурные скобки могут быть опущены.

```
class Empty
```

Open in Playground → Target: JVM Running on v.2.0.0

1.4.3. Свойства класса

Классы в Kotlin могут иметь свойства: изменяемые (mutable) и неизменяемые (read-only) — var и val соответственно.

```
public class Address {
    public var name: String = ...
    public var street: String = ...
    public var city: String = ...
    public var state: String? = ...
    public var zip: String = ...
}
```

Open in Playground → Target: JVM Running on v.2.0.0

Для того, чтобы воспользоваться свойством, мы просто обращаемся к его имени (как в Java):

```
fun copyAddress(address: Address): Address {
    val result = Address() // нет никакого слова new
    result.name = address.name // вызов методов доступа
    result.street = address.street
    // ...
    return result
}
```

Open in Playground → Target: JVM Running on v.2.0.0

1.4.4. Геттеры и сеттеры

Полный синтаксис объявления свойства выглядит так:

```
var <propertyName>: <PropertyType> [= <property_initializer>]
    [<getter>]
    [<setter>]
```

Open in Playground → Target: JVM Running on v.2.0.0

Инициализатор property_initializer, геттер и сеттер можно не указывать. Также необязательно указывать тип свойства, если он может быть выведен из контекста или наследован от базового класса.

Примеры:

```
var allByDefault: Int? // ошибка: необходима явная инициализация,
                       // предусмотрены стандартные геттер и сеттер
var initialized = 1 // имеет тип Int, стандартный геттер и сеттер
```

Open in Playground → Target: JVM Running on v.2.0.0

Синтаксис объявления констант имеет два отличия от синтаксиса объявления изменяемых переменных: во-первых, объявление константы начинается с ключевого слова val вместо var, а во-вторых, объявление сеттера запрещено:

```
val simple: Int? // имеет тип Int, стандартный геттер,
                 // должен быть инициализирован в конструкторе
val inferredType = 1 // имеет тип Int и стандартный геттер
```

Open in Playground → Target: JVM Running on v.2.0.0

Мы можем самостоятельно описать методы доступа, как и обычные функции, прямо при объявлении свойств. Например, пользовательский геттер:

```
val isEmpty: Boolean
    get() = this.size == 0
```

Open in Playground → Target: JVM Running on v.2.0.0

Пользовательский сеттер выглядит примерно так:

```
var stringRepresentation: String
    get() = this.toString()
    set(value) {
        setDataFromString(value) // парсит строку и устанавливает
                                 // значения для других свойств
    }
```

Open in Playground → Target: JVM Running on v.2.0.0

По договорённости, имя параметра сеттера - `value`, но вы можете использовать любое другое.

Если вам нужно изменить область видимости метода доступа или пометить его аннотацией, при этом не внося изменения в реализацию по умолчанию, вы можете объявить метод доступа без объявления его тела:

```
var setterVisibility: String = "abc"
    private set // сеттер имеет private доступ и стандартную реализацию

var setterWithAnnotation: Any? = null
    @Inject set // аннотирование сеттера с помощью Inject
```

Open in Playground → Target: JVM Running on v.2.0.0

1.4.5. Теневые поля

Классы в Kotlin не могут иметь полей. Т.е. переменные, которые вы объявляете внутри класса только выглядят и ведут себя как поля из Java, хотя на самом деле являются свойствами, т.к. для них неявно реализуются методы get и set. А сама переменная, в которой находится значение свойства, называется теневое поле (backing field). Однако, иногда, при использовании пользовательских методов доступа, необходимо иметь доступ к теневому полю. Для этих целей Kotlin предоставляет автоматическое теневое поле, к которому можно обратиться с помощью идентификатора field:

```
var counter = 0
    set(value) {
        if (value >= 0) field = value // значение при инициализации записывается
                                      // прямиком в backing field
    }
```

Идентификатор field может быть использован только в методах доступа к свойству.

Теневое поле будет сгенерировано для свойства, если оно использует стандартную реализацию как минимум одного из методов доступа, либо если пользовательский метод доступа ссылается на него через идентификатор field.

Например, в нижестоящем примере не будет никакого теневого поля:

```
val isEmpty: Boolean
    get() = this.size == 0
```

1.4.6. Теневые свойства

Если вы хотите предпринять что-то такое, что выходит за рамки вышеуказанной схемы "неявного теневого поля", вы всегда можете использовать теневое свойство (backing property):

```
private var _table: Map? = null
public val table: Map     get() {
        if (_table == null) {
            _table = HashMap() // параметры типа вычисляются автоматически
                               // (ориг.: "Type parameters are inferred")
        }
        return _table ?: throw AssertionError("Set to null by another thread")
    }
```

Такой подход ничем не отличается от подхода в Java, так как доступ к приватным свойствам со стандартными геттерами и сеттерами оптимизируется таким образом, что вызов функции не происходит.

1.4.7. Свойства с поздней инициализацией

Обычно, свойства, объявленные non-null типом, должны быть проинициализированы в конструкторе. Однако, довольно часто это неосуществимо. К примеру, свойства могут быть инициализированы через внедрение зависимостей, в установочном методе (ориг.: "setup method") юнит-теста или в методе onCreate в Android. В таком случае вы не можете обеспечить non-null инициализацию в конструкторе, но всё равно хотите избежать проверок на null при обращении внутри тела класса к такому свойству.

Для того, чтобы справиться с такой задачей, вы можете пометить свойство модификатором lateinit:

```
public class MyTest {
    lateinit var subject: TestSubject

    @SetUp fun setup() {
        subject = TestSubject()
    }

    @Test fun test() {
        subject.method()  // объект инициализирован, проверять на null не нужно
    }
}
```

Open in Playground → Target: JVM Running on v.2.0.0

Такой модификатор может быть использован только с var свойствами, объявленными внутри тела класса (не в основном конструкторе, и только тогда, когда свойство не имеет пользовательских геттеров и сеттеров) и, начиная с Kotlin 1.2, со свойствами, расположенными на верхнем уровне, и локальными переменными. Тип такого свойства должен быть non-null и не должен быть примитивным.

Доступ к lateinit свойству до того, как оно проинициализировано, выбрасывает специальное исключение, которое чётко обозначает, что свойство не было определено. Проверка инициализации lateinit var (начиная с версии 1.2)

Чтобы проверить, было ли проинициализировано lateinit var свойство, используйте .isInitialized метод ссылки на это свойство:

```
if (foo::bar.isInitialized) {
    println(foo.bar)
}
```

Open in Playground → Target: JVM Running on v.2.0.0

Эта проверка возможна только для лексически доступных свойств, то есть объявленных в том же типе, или в одном из внешних типов, или глобальных свойств, объявленных в том же файле.

1.4.8. Конструкторы класса

Класс в Kotlin может иметь основной конструктор (primary constructor) и один или более дополнительных конструкторов (secondary constructors). Основной конструктор является частью заголовка класса, его объявление идёт сразу после имени класса (и необязательных параметров):

```
class Person constructor(firstName: String)
```

Open in Playground → Target: JVM Running on v.2.0.0

Если у конструктора нет аннотаций и модификаторов видимости, ключевое слово constructor может быть опущено:

```
class Person(firstName: String)
```

Open in Playground → Target: JVM Running on v.2.0.0

Основной конструктор не может содержать в себе исполняемого кода. Инициализирующий код может быть помещён в соответствующий блок (initializers blocks), который помечается словом init.

При создании экземпляра класса блоки инициализации выполняются в том порядке, в котором они идут в теле класса, чередуясь с инициализацией свойств:

```
class InitOrderDemo(name: String) {
    val firstProperty = "Первое свойство: $name".also(::println)
    init {
        println("Первый блок инициализации: ${name}")
    }
    val secondProperty = "Второе свойство: ${name.length}".also(::println)
    init {
        println("Второй блок инициализации: ${name.length}")
    }
}
fun main() {
    InitOrderDemo("Привет")
}
```

Open in Playground → Target: JVM Running on v.2.0.0

Обратите внимание, что параметры основного конструктора могут быть использованы в инициализирующем блоке. Они также могут быть использованы при инициализации свойств в теле класса:

```
class Customer(name: String) {
    val customerKey = name.toUpperCase()
}
```

Open in Playground → Target: JVM Running on v.2.0.0

В действительности, для объявления и инициализации свойств основного конструктора в Kotlin есть лаконичное синтаксическое решение:

```
class Person(val firstName: String, val lastName: String, var age: Int) {
  // ...
}
```

Open in Playground → Target: JVM Running on v.2.0.0

Свойства, объявленные в основном конструкторе, могут быть изменяемые (var) и неизменяемые (val).

Если у конструктора есть аннотации или модификаторы видимости, ключевое слово constructor обязательно, и модификаторы используются перед ним:

```
class Customer public @Inject constructor(name: String) { ... }
```

Open in Playground → Target: JVM Running on v.2.0.0

Для более подробной информации по данному вопросу см. "Модификаторы доступа". Дополнительные конструкторы

В классах также могут быть объявлены дополнительные конструкторы (secondary constructors), перед которыми используется ключевое слово constructor:

```
class Person {
    var children: MutableList = mutableListOf<>()
    constructor(parent: Person) {
        parent.children.add(this)
    }
}
```

Open in Playground → Target: JVM Running on v.2.0.0

Если у класса есть основной конструктор, каждый дополнительный конструктор должен прямо или косвенно ссылаться (через другой(ие) конструктор(ы)) на основной. Осуществляется это при помощи ключевого слова this:

```
class Person(val name: String) {
    var children: MutableList = mutableListOf<>()
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

Open in Playground → Target: JVM Running on v.2.0.0

Обратите внимание, что код в блоках инициализации фактически становится частью основного конструктора. Дополнительный конструктор ссылается на основной при помощи своего первого оператора, поэтому код во всех блоках инициализации, а также инициализация свойств выполняется перед выполнением кода в теле дополнительного конструктора. Даже если у класса нет основного конструктора на него все равно происходит неявная ссылка и блоки инициализации выполняются также:

```
class Constructors {
    init {
        println("Блок инициализации")
    }
    constructor(i: Int) {
        println("Конструктор")
    }
}
fun main() {
    Constructors(1)
}
```

Open in Playground → Target: JVM Running on v.2.0.0

Если в абстрактном классе не объявлено никаких конструкторов (основного или дополнительных), у этого класса автоматически сгенерируется пустой конструктор без параметров. Видимость этого конструктора будет public. Если вы не желаете иметь класс с открытым public конструктором, вам необходимо объявить пустой конструктор с соответствующим модификатором видимости:

```
class DontCreateMe private constructor () {
}
```

Open in Playground → Target: JVM Running on v.2.0.0

> Примечание: В виртуальной машине JVM компилятор генерирует дополнительный конструктор без параметров в случае, если все параметры основного конструктора имеют значения по умолчанию. Это делает использование таких библиотек, как Jackson и JPA, более простым в языке Kotlin, так как они используют пустые конструкторы при создании экземпляров классов.

```
class Customer(val customerName: String = "")
```

Open in Playground → Target: JVM Running on v.2.0.0

Создание экземпляров классов

Для создания экземпляра класса конструктор вызывается так, как если бы он был обычной функцией:

```
val invoice = Invoice()

val customer = Customer("Joe Smith")
```

Open in Playground → Target: JVM Running on v.2.0.0

Обращаем ваше внимание на то, что в Kotlin не используется ключевое слово new. Классы могут содержать в себе:

- Конструкторы и инициализирующие блоки
- Функции
- Свойства
- Вложенные классы
- Объявления объектов

1.4.9. Анонимные и вспомогательные объекты

Иногда нам необходимо получить экземпляр некоторого класса с незначительной модификацией, желательно без написания нового подкласса. Java справляется с этим с помощью вложенных анонимных классов. Kotlin несколько улучшает данный подход.

Анонимные объекты (ориг.:Object expressions)

Для того, чтобы создать объект анонимного класса, который наследуется от какого-то типа (типов), используется конструкция:

```
window.addMouseListener(object : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) {
        // ...
    }
    override fun mouseEntered(e: MouseEvent) {
        // ...
    }
})
```

Open in Playground → Target: JVM Running on v.2.0.0

Если у супертипа есть конструктор, то в него должны быть переданы соответсвующие параметры. Множество супертипов может быть указано после двоеточия в виде списка, заполненного через запятую:

```
open class A(x: Int) {
    public open val y: Int = x
}
interface B {...}
val ab: A = object : A(1), B {
    override val y = 15
}
```

Open in Playground → Target: JVM Running on v.2.0.0

Если всё-таки нам нужен просто объект без всяких там родительских классов, то можем указать:

```
val adHoc = object {
    var x: Int = 0
    var y: Int = 0
}
print(adHoc.x + adHoc.y)
```

Open in Playground → Target: JVM Running on v.2.0.0

Код внутри объявленного объекта может обращаться к переменным за скобками так же, как вложенные анонимные классы в Java

```
fun countClicks(window: JComponent) {
    var clickCount = 0
    var enterCount = 0
    window.addMouseListener(object : MouseAdapter() {
        override fun mouseClicked(e: MouseEvent) {
            clickCount++
        }
        override fun mouseEntered(e: MouseEvent) {
            enterCount++
        }
    })
    // ...
}
```

Open in Playground → Target: JVM Running on v.2.0.0

Объявления объектов (ориг.:Object declarations)

Синглтон - очень полезный паттерн программирования, и Kotlin (переняв у Scala) позволяет объявлять его довольно простым способом :

```
object DataProviderManager {
    fun registerDataProvider(provider: DataProvider) {
        // ...
    }
    val allDataProviders: Collection
        get() = // ...
}
```

Open in Playground → Target: JVM Running on v.2.0.0

Это называется объявлением объекта и всегда имеет приставку в виде ключевого слова object. Аналогично объявлению переменной, объявление объекта не является выражением и не может быть использовано в правой части оператора присваивания.

Для непосредственной ссылки на объект используется его имя:

```
DataProviderManager.registerDataProvider(...)
Подобные объекты могут иметь супертипы:
object DefaultListener : MouseAdapter() {
```

```
    override fun mouseClicked(e: MouseEvent) {
        // ...
    }
    override fun mouseEntered(e: MouseEvent) {
        // ...
    }
}
```

Open in Playground → Target: JVM Running on v.2.0.0

Объявление объекта не может иметь локальный характер (т.е. быть вложенным непосредственно в функцию), но может быть вложено в объявление другого объекта или какого-либо не вложенного класса.

Вспомогательные объекты

Объявление объекта внутри класса может быть отмечено ключевым словом companion:

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}
```

Open in Playground → Target: JVM Running on v.2.0.0

Для вызова членов такого companion объекта используется имя класса:

```
val instance = MyClass.create()
```

Open in Playground → Target: JVM Running on v.2.0.0

Не обязательно указывать имя вспомогательного объекта. В таком случае он будет назван Companion:

```
class MyClass {
    companion object {
    }
}
val x = MyClass.Companion
```

Open in Playground → Target: JVM Running on v.2.0.0

Такие члены вспомогательных объектов выглядят, как статические члены в других языках программирования. На самом же деле, они являются членами реальных объектов и могут реализовывать, к примеру, интерфейсы:

```
interface Factory {
    fun create(): T
}
class MyClass {
    companion object : Factory {
        override fun create(): MyClass = MyClass()
    }
}
```

Open in Playground → Target: JVM Running on v.2.0.0

Однако в JVM вы можете статически генерировать методы вспомогательных объектов и полей, используя аннотацию @JvmStatic@. См. Совместимость с Java. Семантическое различие между анонимным объектом и декларируемым объектом.

Существует только одно смысловое различие между этими двумя понятиями:

- анонимный объект инициализируется сразу после того, как был использован
- декларированный объект инициализируется лениво, в момент первого к нему доступа
- вспомогательный объект инициализируется в момент, когда класс, к которому он относится, загружен и семантически совпадает со статическим инициализатором Java