Объектно-ориентированное программирование в Kotlin

Объектно-ориентированное программирование в Kotlin имеет ряд особенностей, связанных с появлением первичного конструктора, у полей – геттеров и сеттеров по-умолчанию. Есть возможность определять методы за пределами класса.

В Kotlin по-умолчанию классы и их функции имеют модификатор `final`. Это значит, что от таких классов нельзя наследовать, а функции нельзя переопределять в дочерних классах. Чтобы разрешить наследование и переопределение необходимо это указать явно – писать модификатор `open` у родительского класса и его методов. В дочернем классе переопределяемые методы должны иметь модификатор `override`.

В Kotlin есть абстрактные классы и интерфейсы, они являются открытыми по-умолчанию.

Дочерние классы могут наследовать от одного родительского и от множества интерфейсов.

```
open class Parent
interface First
interface Second

class Child: Parent(), First, Second
```

Поскольку в Kotlin параметры функций могут иметь значения по-умолчанию, то зачастую в классах достаточно одного конструктора, который обычно делают первичным и нередко объединяют с объявлением свойств.

```
fun main() {
    val obj1 = MyClass()
    val obj2 = MyClass(10, 5)
}
```

```
class MyClass(var a: Int = 0, var b: Int = 0)
```

Приведенное выше определение *MyClass* это сокращенный вариант от примерно такого:

```
class MyClass(aa: Int = 0, bb: Int = 0) {
    var a = aa
    var b = bb
}
```

Если преобразовать первичный конструктор ко вторичному, определение класса получится более характерным для других языков программирования, но зато и более длинным:

```
class MyClass {
    var a: Int
    var b: Int

    constructor(aa: Int = 0, bb: Int = 0) {
        a = aa
        b = bb
    }
}
```

В случае необходимости у класса может быть несколько конструкторов, среди которых только один первичный. При наличии первичного конструктора вторичные должны делегировать к нему непосредственно или опосредованно через другой вторичный. Делается это с помощью ключевого слова `this`.

```
fun main() {
    val obj1 = MyClass()
    println(obj1.d) // true
    val obj2 = MyClass(10, 5)
    println(obj2.d) // false
}
```

```
class MyClass(aa: Int, bb: Int) {
    var a = aa
    var b = bb
    var d = false

    constructor(): this(0, 0) {
        d = true
    }
}
```

Поскольку первичный конструктор не имеет тела, в случае необходимости какого-либо инициирующего кода, он заключается в блок `init{}`.

```
fun main() {
    val obj1 = MyClass()
    println(obj1.d) // true
    val obj2 = MyClass(0, 0)
    println(obj2.d) // true
}

class MyClass(aa: Int = 0, bb: Int = 0) {
    var a = aa
    var b = bb
    var d: Boolean
    init {
        if (aa == 0 && bb == 0)
            d = true
        else d = false
    }
}
```

Каждое поле класса имеет свои геттер и сеттер, вместе они формируют свойство. По-умолчанию геттер возвращает значение поля и сеттер присваивает полю значение без всякой обработки. Явно это выражается так:

```
class MyClass(aa: Int, bb: Int) {
    var a = aa
        get() {return field}
```

```
        set(value) {field = value}

    var b = bb
        get() {return field}
        set(value) {field = value}
}
```

Слово `field` обозначает текущее поле. Используется, чтобы избежать рекурсивных вызовов. Явное указание геттеров и сеттеров имеет смысл, если их логика сложнее, или надо запретить возможность изменения значения поля за пределами класса, оставив возможность получать значение.

```
fun main() {
    val obj1 = MyClass(12, 3)
    obj1.a = -5
    println(obj1.a) // 12
    // obj1.b = 9 Error
    println(obj1.b) // Ok 3
}

class MyClass(aa: Int, bb: Int) {
    var a = aa
        set(value) {
            if ((value) > 0)
                field = value
        }

    var b = bb
        private set
}
```

В Kotlin методы, определенные внутри класса, называются функциями-членами. Функции-расширения – это методы, определенные за пределами класса.

```
fun main() {
    val obj1 = MyClass(12, 3)
    obj1.member()
    obj1.extension()
```

```
}

class MyClass(val a: Int = 0, val b: Int = 0) {

    fun member() {
        println("member-function")
        println(a + b)
    }
}

fun MyClass.extension() {
    println("extension-function")
    println(a - b)
}
```

В Kotlin многие методы встроенных классов реализованы как функции-расширения. Это связано с тем, что Kotlin во многом использует java-библиотеки, и разработчикам было проще дополнить их код за пределами классов.

Функции-расширения позволяют расширять возможности уже имеющихся классов (в том числе из стандартных и сторонних библиотек), не внося изменения в их код и программы, которые ранее использовали эти классы без расширений.

PDF-версия курса с дополнительными уроками