Совместное использование языков Kotlin и Java

Романов Владимир Юрьевич

Декабрь 19, 2020

5. СОВМЕСТНОЕ ИСПОЛЬЗОВАНИЕ ЯЗЫКОВ KOTLIN И JAVA

5.1 ВЫЗОВ КОДА JAVA ИЗ KOTLIN

Геттеры и сеттеры

- *Геттеры* и *сеттеры* в **Java**
  - *Геттер* - метод без аргументов с именем начинающимися на ***get***
  - *Сеттер* - метод с единственным аргументом, имя которого начинаются на ***set***
- *Геттеры* и *сеттеры* представлены в **Kotlin** как ***свойства***.

```
import java.util.Calendar

fun calendarDemo() {
 val calendar = Calendar.getInstance()

 // Вызов getFirstDayOfWeek()
 if (calendar.firstDayOfWeek == Calendar.SUNDAY) {
    // Вызов setFirstDayOfWeek()
    calendar.firstDayOfWeek = Calendar.MONDAY
 }
}
```

Методы возвращающие void

- Если метод **Java** возвращает **void**, он вернет **Unit** при вызове из **Kotlin**.
- Если по какой-либо причине кто-либо использует данный тип в своих целях, он будет присвоен в месте вызова компилятором **Kotlin**, так как значение само по себе известно заранее (являясь **Unit**).

Геттеры и сеттеры (**Boolean**)

- Методы доступа к полям типа **Boolean**
  - Имя геттера начинается с ***is***
  - Имя сеттера начинается с ***set***
- Методы доступа к полям типа **Boolean** в **Kotlin** как ***свойства***, у которых *имя совпадает с именем геттера*.

```
import java.util.Calendar

fun calendarDemo() {
    val calendar = Calendar.getInstance()
    // Вызов isLenient()
    if (!calendar.isLenient) {
        // Вызов setLenient()
        calendar.isLenient = true
    }
}
```

Экранирование идентификаторов **Java**

- **is** в языке **Kotlin** - это ключевое слово
- **is** в языке **Java** может быть идентификатором
- экранирование с помощью обратной кавычки (`) может применяться для использования таких идентификаторов в программе на языке **Kotlin**

```
foo.`is`(bar)
```

Нулевые ссылки из Java

- Любая ссылка в **Java** может прнимать значение **null**
- Поэтому объекты приходящие из **Java** потенциально опасны
- Типы, декларируемые в **Java** обрабатываются в **Kotlin** по-особому
- Такие типы называются *платформенными типами*.
- Проверки на **null** для платформенных типов являются менее строгими,
- Безопасность для платформенных типов гарантирована также как в как в **Java**.

Платформенные типы из Java (пример)

```
val list = ArrayList<String>() // non-null (результат выполнения конструктора)

list.add("Item")

val size = list.size // non-null (примитив int)

val item = list[0] // подразумевается платформенный тип (обычный объект Java)

item.substring(1) // разрешается, может выбросить исключение, если item == null

val nullable: String? = item // разрешается, всегда работает

val notNull: String = item // разрешается, может вызвать ошибку на рантайме
```

Нотации для платформенных типов

- Платформенные типы не могут быть явно указаны в программе, так что не существует синтаксиса для их обозначения в языке.
- Компилятору и IDE нужно их обозначать (в сообщениях об ошибках, информации о параметрах, и т.д.).
- Для подобных случаев у нас имеется мнемоническая нотация для них:
- T! означает T или T?

- (Mutable)Collection<T>! означает *коллекция* **Java** типа **T** может быть (не)изменяемой, может быть **nullable** или нет
- Array<(out) T>! означает *массив **Java** типа **T** (или подтипа **T**), **nullable** или нет*

Аннотации допустимости null значений

- Типы Java, которые имеют *аннотации допустимости* **null** - значений представлены в языке **Kotlin** не как *платформенные типы*, а как реальные **nullable** или **non-null** типы **Kotlin**.
- Компилятор **Kotlin** поддерживает несколько стандартов аннотаций для обозначения **null** значений:
  - **JetBrains** (@Nullable и @NotNull из пакета org.jetbrains.annotations)
  - **Android** (com.android.annotations и android.support.annotations)
  - **JSR-305** (javax.annotation)
  - **FindBugs** (edu.umd.cs.findbugs.annotations)
  - **Eclipse** (org.eclipse.jdt.annotation)
  - **Lombok** (lombok.NonNull)

Родовые типы **Java** в **Kotlin**

Родовые типы **Kotlin** имеют отличия от **Java**. При импорте родовых типов **Java** в **Kotlin** выполняютя преобразования:

Подстановочные знаки **Java** преобразуются в проекции типов,

- Foo<? extends Bar> становится Foo<out Bar!>!
- Foo<? super Bar> становится Foo<in Bar!> !

Исходные типы **Java** преобразуются в *проекции с символом* *

- List становится List<*>!, т.е. List<out Any?>!

Родовые типы **Java** в **Kotlin** во время выполнения

- Как и **Java**, родовые типы **Kotlin** не сохраняются во время выполнения
- Объекты не хранят информацию о фактических аргументах типа, переданных их конструкторам
- Пример: ArrayList<Integer>() неотличим от ArrayList<Character>()
- По этой причине нельзя выполнить проверку для *родовых типов* с помощью оператора ***is***

В **Kotlin** допускается только проверки для родовых типов *проекции с символом* *

```
// Ошибка: не удается проверить,
// действительно ли это список целых
if (a is List<Int>)

// НЕ ошибка: не требуется никаких гарантий
// относительно содержимого списка
if (a is List<*>)
```

Массивы Java в Kotlin

- Массивы в **Kotlin** *инвариантны*
- Массивы в **Java** не *инвариантны*
- В **Kotlin** нельзя переменной типа Array<String> присвоить значение типа Array<Any>
- Это предотвращает возможный сбой во время выполнения.

Методы Java с переменным числом параметров

- В ***Java*** иногда используюся объявления методов с переменным числом аргументов (***varargs**):

```
public class JavaArrayExample {
    public void removeIndicesVarArg(int... indices) {
        // ...
    }
}
```

Для преобразования массива в список параметров нужно использовать оператор *:

```
val javaObj = JavaArrayExample()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndicesVarArg(*array)
```

Проверяемые (checked) исключения

- В **Kotlin** все исключения непроверяемые (*unchecked*),
- Компилятор не будет заставлять их обрабатывать.
- При вызове **Java** метода, который может выбросить проверяемое (*checked*) исключение, компилятор **Kotlin** не заставит Вас его обработать:

```
fun render(list: List<*>, to: Appendable) {
    for (item in list) {
        to.append(item.toString())
        // Java потребует здесь обработки IOException
    }
}
```

SAM Преобразования

- Как и **Java8**, **Kotlin** поддерживает **SAM** (*Single Abstract Method*) преобразования.
- литералы функций в **Kotlin** могут быть автоматически преобразованы к реализации **Java** интерфейсов с одним абстрактным методом, если типы параметров функции в **Kotlin** совпадают с типами параметров метода интерфеса.

```
val runnable = Runnable { println("This runs in a runnable") }
val executor = ThreadPoolExecutor()
// Java signature: void execute(Runnable command)
executor.execute { println("This runs in a thread pool") }
```

Доступ к статическим членам класса

- Статические члены классов **Java** образуют *объекты-дополнения* (компаньоны) для этих классов.
- Невозможно передать такой *объект-компаньон* как значение
- Можно получить доступ к членам явно, например:

```
if (Character.isLetter (a)) {
    // ...
}
```

Использование JNI в Kotlin

Чтобы объявить функцию, реализованную на нативном (**C** или **C++**) коде, нужно пометить ее модификатором **external**:

```
external fun foo(x: Int): Double
```

В остальном, процедура работы аналогична **Java**.

5.2 ВЫЗОВ КОДА KOTLIN ИЗ JAVA

Свойства (Properties)

Свойство **Kotlin** компилируется в следующие **Java** элементы:

- Метод *getter*, имя которого вычисляется путем добавления префикса **get**
- Метод *setter*, имя которого вычисляется путем добавления префикса **set** (только для свойств **var**);
- Приватное поле с тем же именем, что и имя свойства (только для свойств с *backing fields*).
- Например, var firstName: String компилируется в следующие объявления **Java**:

```
private String firstName;

public String getFirstName () {
    return firstName;
}

public void setFirstName (String firstName) {
    this.firstName = firstName;
}
```

Свойства (Properties) начинаются на **is**

- Если имя свойства начинается с **is**, используется другое правило сопоставления имен:
  - имя метода *getter* будет совпадать с именем свойства,
  - имя метода *setter* будет получено путем замены **is** на **set**.
- Например, для свойства **isOpen**,
  - *getter* будет называться **isOpen()**
  - *setter* будет называться **setOpen()**
- Это правило применяется к свойствам любого типа, а не только к **Boolean**

Функции уровня пакета (*Package-Level Functions*)

Все функции и свойства, объявленные в файле **example.kt** внутри пакета **org.foo.bar**, включая функции расширения, скомпилированы в статические методы класса **Java** с именем **org.foo.bar.ExampleKt**.

```
// example.kt
package demo

class Point

fun draw() { ... }
```

```
// Java
new demo.Point();
demo.ExampleKt.draw();
```

Смена имени генерируемого Java класса

Имя генерируемого **Java** класса может быть выбранно при помощи аннотации ***@JvmName***:

```
@file:JvmName("DemoUtils")

package demo

class Foo

fun bar() { ... }
// Java
new demo.Foo();
demo.DemoUtils.bar();
```

Генерация класса-фасада на языке Java

```
// oldutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass
package demo

fun foo() { ... }

// newutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass
package demo

fun bar() { ... }
// Java
demo.Utils.foo();
demo.Utils.bar();
```

Поля экземпляра (Instance Fields)

- Для использования свойство **Kotlin** в качестве поля в **Java**, нужно добавить к нему аннотацию ***@JvmField***.
- Поле будет иметь такую же видимость, что и базовое свойство.
- Можно добавить свойству аннотацию ***@JvmField***, если
  - оно имеет *backing field*
  - не является приватным,
  - не имеет *модификаторов* **open**, **override** или **const**
  - не является свойством-делегатом

```
class C(id: String) {
    @JvmField val ID = id
}
// Java
class JavaClient {
    public String getID(C c) { return c.ID; }
}
```

Статические поля (Static Fields)

- Свойства **Kotlin**, объявленные в именованном объекте (*object*) или *объекте-companion*, будут иметь статические *backing fields* в этом именованном объекте или в классе, содержащем *объект-companion*.
- Обычно эти поля являются приватными, но они могут быть представлены одним из следующих способов:
  - ***@JvmField*** аннотацией
  - ***lateinit*** модификатором
  - ***const*** модификатором
- Аннотирование такого свойства с помощью **@JvmField** делает его статическим полем с той же видимостью, что и само свойство

Статические поля (Static Fields) пример

```
class Key(val value: Int) {
    companion object {
        @JvmField
        val COMPARATOR: Comparator<Key>
                = compareBy<Key> { it.value }
    }
}
// Java
Key.COMPARATOR.compare(key1, key2);
// public static final field in Key class
```

Константы верхнего уровня и классов

```
// file example.kt
object Obj {
    const val CONST = 1
}
class C {
    companion object {
        const val VERSION = 9
    }
}

const val MAX = 239
// Java
int c = Obj.CONST;
int d = ExampleKt.MAX;
int v = C.VERSION;
```

Статические методы

- В **Kotlin** представляются функции уровня пакета как статические методы.
- В **Kotlin** также может генерировать статические методы для функций, определенных в именованных объектах или объектах-companion, если добавляется аннотация **@JvmStatic** к функции.
- Если используется эта аннотация, компилятор создаст как статический метод во включающем классе объекта, так и метод экземпляра в самом объекте.

Статические методы. Пример

```
class C {
    companion object {
        @JvmStatic fun draw() {}
        fun bar() {}
    }
}
```

Теперь **draw()** является статическим методом в **Java**, **bar()** не является статическим методом в **Java**:

```
C.draw(); // Работает
C.bar(); // Ошибка: не статический метод
C.Companion.draw(); // Метод экземпляра
C.Companion.bar(); // Единственный способ вызова
```

Методы именованных объектов

```
object Obj {
    @JvmStatic fun foo() {}
    fun bar() {}
}
```

В **Java**:

```
Obj.foo(); // Работает
Obj.bar(); // Ошибка
Obj.INSTANCE.bar(); // Работает как вызов экземпляра singleton
Obj.INSTANCE.foo(); // Также работает
```

Видимость

Видимость в **Kotlin** представляется в **Java** следующим образом:

- **private** элементы компилируются в **private** элементы
- **private** объявления верхнего уровня компилируются в локальные объявления пакетов;
- **protected** остаются **protected**.
  Java разрешает доступ к защищенным членам из других классов в том же пакете, а **Kotlin** - не разрешает.
  Поэтому классы **Java** будут иметь более широкий доступ к коду
- **internal** объявления становятся **public** в **Java**.
- **public** остаются **public**.

Вызов метода Kotlin с параметром типа KClass.

- Автоматического преобразования из **Class** в **KClass** нет
- Нужно сделать это вручную, вызывая эквивалент свойства расширения **Class.kotlin**:

```
kotlin.jvm.JvmClassMappingKt.getKotlinClass(MainView.class)
```

Проверенные исключения (Checked Exceptions)

- **Котлин** не имеет проверенных исключений.
- Сигнатуры **Java** функций **Kotlin** не объявляют исключения. Поэтому если мы имеем такую функцию в Котлине:

```
// example.kt
package demo

fun draw() {
    throw IOException()
}
```

И мы хотим вызвать её из **Java** и поймать исключение:

```
// Java
try {
  demo.Example.draw();
}
catch (IOException e) { // error: draw() does not declare IOException in the throws list
  // ...
}
```

мы получаем сообщение об ошибке из компилятора **Java**, потому что **draw()** не объявляет **IOException**.

Проверенные исключения (Checked Exceptions)

Чтобы обойти эту диагностику, используется аннотация **@Throws** в **Kotlin**:

```
@Throws(IOException::class)
fun draw() {
    throw IOException()
}
// Java
try {
  demo.Example.draw();
}
catch (IOException e) {
  // ...
}
```