

# Язык программирования Kotlin

Андрей Бреслав — разработчик языка Kotlin, компания JetBrains

**Ключевые слова:** язык программирования, JVM, ООП, Kotlin, Java

## Аннотация

В последние годы назрела потребность в новом языке, компилируемом в переносимый байт-код для виртуальной машины Java. В результате появилось несколько проектов, один из которых — Kotlin, статически типизированный объектно-ориентированный язык, совместимый с Java и предназначенный для промышленной разработки приложений.

## Введение

История разработки “альтернативных” языков на платформе Java насчитывает более десятилетия, однако распространения такие языки добились относительно недавно. Так, на волне популярности динамически типизированных языков поднялись JRuby [], Groovy [] и Clojure []. Среди статически типизированных языков следует упомянуть Scala [], Fantom [] и Gosu []. Сам язык Java тоже не стоит на месте, но его развитие осложнено как необходимостью сохранения обратной совместимости, так и непростой судьбой компании Sun Microsystems, поглощенной Oracle в 2009 году.

На этом фоне новый проект компании JetBrains под кодовым названием “Kotlin” (ударение на “o”) с одной стороны выглядит “данью моде”, а с другой — находится в окружении заметного числа конкурентов, однако мы чувствуем себя достаточно уверенно в этой ситуации, по нескольким причинам. Во-первых, JetBrains уже более десяти лет успешно занимается интегрированными средами разработки для разных языков программирования (многие из которых работают на платформе Java), и за это время была собрана сильная команда специалистов и накоплен значительный опыт в области языков программирования и смежных технологий. Во-вторых, мы не можем сказать, что какой-либо из существующих языков на платформе Java, удовлетворяет наши потребности в новом языке, и полагаем (как показывают предварительные отзывы, которые мы получаем от программистов со всего мира, полагаем небезосновательно), что наши коллеги в других компаниях испытывают похожие затруднения.

Итак, летом 2010 года началась разработка проекта. Были сформулированы следующие требования к будущему языку: он должен быть

- **совместим с Java “в обе стороны”**: код на Java можно вызывать из кода на Kotlin, и наоборот;

- **компилироваться как минимум так же быстро как Java**, это требование важно для больших проектов, таких, например, как проект IntelliJ IDEA;
- **быть безопаснее Java**, то есть статически гарантировать отсутствие ошибок, типичных для программ на Java;
- **быть лаконичнее Java**. Всем известны обвинения Java в излишней “церемониальности”: код на этом языке изобилует “само собой разумеющимися” конструкциями, загромождающими программы;
- и, наконец, при сохранении необходимой выразительности, новый язык должен быть **значительно проще Scala**, поскольку сложность освоения — очень существенный фактор.

В июле 2011 он был официально анонсирован и на сайте <http://jetbrains.com/kotlin> размещено описание языка. Выпуск публичной бета-версии компилятора запланирован на конец 2011 года.

В настоящей статье мы расскажем об основных особенностях языка Kotlin и приведем несколько примеров его использования.

## Основные элементы языка

**Функции.** Kotlin — объектно-ориентированный язык, но, в отличие от Java, он позволяет объявлять функции вне классов. В Java для этих целей используются статические методы, что приводит к возникновению классов, фактически не являющихся таковыми: объекты этих классов никогда не создаются, и вызываются лишь статические методы.

Объявления в Kotlin объединяются в *пространства имен* (namespaces), и функция может быть объявлена непосредственно внутри пространства имен:

```
namespace example {
    fun max(a : Int, b : Int) : Int {
        if (a < b)
            return a
        return b
    }
}
```

Как видно из примера, объявление функции предваряется ключевым словом `fun`, а типы параметров, указывается после двоеточия, следующего за именем параметра. Аналогично обозначается тип возвращаемого значения функции. Такой синтаксис следует традициям языка ML и, в частности, Scala. Он позволяет легко опускать типовые аннотации, если тип может быть *выведен* компилятором автоматически, из контекста.

**Переменные** в Kotlin, как и в Scala, объявляются с помощью ключевых слов `val` (неизменяемые переменные) и `var` (изменяемые):

```

var sinSum : Double = 0.0
for (x in xs) {
    val y = syn(x)
    sinSum += y
}

```

В данном примере тип неизменяемой переменной `y` опущен, поскольку компилятор может его определить из значения правой части определения переменной. Тип изменяемой переменной `sinSum` указан явно лишь для того, чтобы продемонстрировать синтаксис: компилятору достаточно информации, чтобы вывести тип и в этом случае.

**Классы.** Основным инструментом декомпозиции в Kotlin, как и других ОО-языках, являются классы. При объявлении класса список параметров конструктора указывается непосредственно в заголовке:

```
class IntPair(x : Int, y : Int) { ... }
```

Экземпляры класса создаются прямым вызовом конструктора; ключевого слова `new` в Kotlin нет:

```
val xy = IntPair(x, y)
```

В заголовке класса также указывается *список супертипов*, отделяемый двоеточием:

```
class MyList<E>(length : Int) : List<E>, Serializable { ... }
```

**Трейты.** Класс может наследоваться от одного класса и/или от нескольких *трейтов* (traits). Трейты похожи на классы тем, что они тоже определяют типы, тоже могут иметь функции-члены и наследоваться от других трейтов. Основное отличие состоит в том, что трейты не имеют конструкторов и, как следствие, не могут иметь *состояния* (полей) и . Можно сказать, что трейты — это знакомые всем интерфейсы из языка Java, только функции в них могут иметь реализацию. Ограничения, накладываемые на трейты, позволяют избежать трудностей, связанных с *множественным наследованием классов*.

**Внешние функции.** Еще один механизм “расширения” типов в Kotlin — это внешние функции (extension functions, “функции-расширения”). Такие функции могут быть объявлены вне какого-либо класса и при этом вызываться так как будто они были объявлены внутри. Вот пример объявления внешней функции для типа `Int`:

```

fun Int.abs() : Int {
    if (this < 0) return -this
    else return this
}

```

```
}
```

Тип, расширяемый данной функцией, указывается перед ее именем и отделяется точкой. Это соответствует объявлению “неявного” параметра, который внутри функции обозначается ключевым словом `this`. Такую функцию можно вызывать с помощью операции “точка”, как и функции-члены класса:

```
val x = (-1).abs()
```

Такой синтаксис позволяет реализовывать в классах лишь необходимый минимум функциональности без вреда для читаемости программы.

Отметим, что внешние функции связываются *статически*, то есть не являются виртуальными (*virtual*).

## Система типов

**Нулевые ссылки.** Система типов языка Kotlin позволяет гарантировать отсутствие в программах некоторых видов ошибок, например, разадресации нулевой ссылки. Типы в Kotlin делятся на содержащие `null` (`Nullable types`) и не содержащие `null` (`Non-null types`). Типы, содержащие `null`, аннотируются вопросительным знаком:

```
fun isEmpty(s : String?) : Int { ... }
```

Знак вопроса обозначает, что ссылка `s` указывает на объект класса `String` *или имеет значение `null`*. Результат функции `isEmpty`, в свою очередь, не может иметь значения `null`, и должен быть целым числом. Компилятор не позволяет разадресовывать ссылку типа `String?` без предварительной проверки:

```
return s.length() == 0 // Ошибка: разадресация нулевой ссылки
```

Необходимо явно проверить, ссылается ли `s` на существующий объект:

```
if (s != null) {  
    return s.length() == 0 // s гарантированно ссылается на существующий объект  
} else {  
    return true  
}
```

или

```
return (s == null) || s.length() // Семантика оператора || обеспечивает проверку
```

Часто встречаются длинные цепочки вызовов, каждый из которых может вернуть `null`. В результате мы получаем несколько вложенных условий,

проверяющих, что вернул каждый из вызовов в цепочке. Чтобы избежать загромождения кода, в Kotlin поддерживается оператор *безопасного вызова*, обозначаемый “?.”:

```
a?.getB()?.getC()?.getD()
```

Если *a* не равно null, выражение *a?.getB()* возвращает *a.getB()*, а в противном случае — null.

**Автоматическое приведение типа.** Выше мы приводили пример того, как компилятор учитывает информацию, содержащуюся в условиях, и разрешает разыменовывать уже проверенные ссылки. Аналогичный механизм автоматически вставляет *приведения типа*, если выше проверялось соответствующее условие. Оператор проверки типа (аналог *instanceof*) в Kotlin называется *is*:

```
val x : Any = ...
if (x is String) {
    print(x.length())
}
```

В этом примере ссылка *x* проверяется на принадлежность к типу *String*, и если проверка прошла успешно, на экран выводится длина строки. При вызове функции *length()* компилятор автоматически вставляет приведение *x* к типу *String*, поскольку оно безопасно в этом месте программы.

**Обобщенные типы, вариантность.** Обобщенные типы (*generics*) необходимы для описания типизированных коллекций (именно для этой цели они были введены в C# 2.0 и в Java 5), однако имеют много других весьма важных применений. В отличие от Java, в Kotlin поддерживается *вариантность* типовых параметров.

Проблему вариантности проще всего проиллюстрировать следующим примером. Пусть объявлен класс *Producer<T>*, имеющий единственный метод *produce*:

```
class Producer<T>() {
    fun produce() : T { ... }
}
```

В случае Java, переменной типа *Producer<Object>* нельзя присвоить ссылку на объект типа *Producer<String>*, поскольку эти типы *несравнимы*. В целом ряде случаев такое поведение компилятора оправдано, ярким примером служат изменяемые коллекции, для которых такое присваивание может привести к ошибкам времени выполнения (см. [1]). Однако в случае нашего класса *Producer* это было бы безопасно, поскольку параметр *T* только *возвращается* из методов этого класса и никогда не *принимается* в качестве параметра. В таких случаях

класс называется *ковариантным* по параметру T. В Java эта проблема разрешается на стороне использования с помощью “немых типовых переменных”, например `Producer<? extends Object>`, что приводит к трудночитаемым ошибкам компиляции и загромождает код.

В Kotlin, чтобы сообщить компилятору о том, что параметр T является ковариантным, он помечается словом `out`, после чего его свойства будут учитываться при проверке типов: С одной стороны компилятор убедится и том, что внутри класса мы только возвращаем T (и выдаст сообщение об ошибке, если это не так). С другой стороны, тип `Producer<Child>` будет считаться подтипом `Producer<Parent>`, если A является подтипом B, и соответствующие присваивания будут разрешены:

```
class Producer<out T>() {  
    fun produce() : T { ... }  
    // fun consume(t : T) { ... } — такой метод в этом классе объявить нельзя  
}
```

```
val p : Producer<Object> = Producer<String>()
```

Кроме `out`-параметров, Kotlin поддерживает `in`-параметры, называемые *контравариантными* и работающие “наоборот”: методы класса (назовем его `Consumer`) могут только принимать их на вход, и не могут возвращать, а `Consumer<Parent>` является подтипом `Consumer<Child>`.

Если параметр не помечен как `in` или `out`, он считается *инвариантным*, то есть ведет себя как в Java.

## Функции высших порядков

Значительное число паттернов объектно-ориентированного проектирования (см [1]) так или иначе сводятся к использованию *стратегии* (*strategy*). Этот основополагающий паттерн позволяет писать код, параметризованный обращениями к некоторому интерфейсу. Например, для того, чтобы реализовать метод сортировки, работающий для объектов произвольного типа, необходимо параметризовать его объектом, который “умеет” сравнивать значения этого типа и отвечать на вопрос, какое из них меньше. Этот объект является стратегией сравнения или *компаратором* (*comparator*).

Говоря о стратегиях, мы используем терминологию ООП, однако очень схожий подход является основополагающим в *функциональном программировании*: так, продолжая пример с методом сортировки, для того, чтобы отсортировать набор значений произвольного типа достаточно написать функцию, которая принимает в качестве параметра другую функцию — функцию сравнения для элементов этого типа. Функция сортировки является, таким образом, *функцией высшего порядка*, поскольку принимает другую функцию в качестве параметра.

Итак, знакомый всем ОО-программистам паттерн Стратегия оказывается популярным и функциональном мире. Важным отличием является то, что в функциональных языках эта техника реализуется гораздо менее многословно. Так, например, в классическом C++ для создания объекта-стратегии требуется объявить именованный класс, что сильно загромождает код. В Java можно использовать анонимные классы, которые к тому же могут обращаться к локальным переменным в лексической области видимости, что облегчает использование стратегий, но все же требует написания довольно значительного количества “технического” кода, не несущего смысловой нагрузки. В функциональных же языках программирования достаточно использовать так называемые *лямбда-выражения* или *функциональные литералы*, не требующие громоздких объявлений.

Являясь объектно-ориентированным языком, Kotlin тем не менее поддерживает функции высших порядков. Это означает, в первую очередь, что функция в Kotlin может являться значением, имеющим соответствующий тип:

```
val intLess : fun(x : Int, y : Int) : Boolean = ...
```

В данном примере переменная `intLess` имеет *функциональный тип* “`fun(x : Int, y : Int) : Boolean`”, то есть хранимое ею значение является функцией, принимающей два целочисленных параметра и возвращающих булево значение (из имени переменной следует, что функция возвращает `true`, если `x` меньше `y`). В частности, мы можем вызвать эту функцию:

```
val less = intLess(1, 2)
```

Такая функция может использоваться при сортировке набора целых чисел:

```
fun intSort(ints : Collection<Int>, intLess : fun(x : Int, y : Int) : Boolean) : List<Int> { ... }
```

В общем случае можно использовать обобщенные типы:

```
fun sort<T>(coll : Collection<T>, intLess : fun(x : T, y : T) : Boolean) : List<T> { ... }
```

Самое интересное — как задавать значение аргумента, имеющего функциональный тип. Для этого используются уже упоминавшиеся выше *функциональные литералы*:

```
val sortedDesc = sort(ints, {x, y => x > y})
```

Второй аргумент представляет собой короткое объявление функции: оно обязательно заключается в фигурные скобки, до символа “`=>`” следуют объявления параметров, а после — тело функции, при этом оператор `return` не требуется, поскольку результатом считается *последнее выражение* в теле

литерала. Таким образом, в приведенном примере, коллекция `ints` будет отсортирована *по убыванию*, поскольку переданная нами функция возвращает `true`, когда `x` больше `y`.

В приведенном примере типы параметров функционального литерала не указаны, поскольку компилятор автоматически выведет их из контекста. При необходимости типы можно указать, но в большинстве случаев такая короткая запись возможна и позволяет сделать код значительно более читабельным.

Приведем еще один пример. Функция `lock` принимает два параметра: объект синхронизации (монитор) и функцию, во время выполнения сначала захватывается монитор, затем в блоке `try..finally` выполняется функция, а затем монитор освобождается (тип `Unit` примерно соответствует `void` в C и Java):

```
fun lock(l : Lock, body : fun() : Unit) {  
    l.acquire()  
    try {  
        body()  
    } finally {  
        l.release()  
    }  
}
```

Для того, чтобы вызвать фрагмент кода с синхронизацией, достаточно написать следующее:

```
lock(myLock, {  
    // Код, который необходимо выполнить  
})
```

Для упрощения использования подобных функций в Kotlin принята следующая *конвенция*: если последним аргументом при вызове функции является функциональный литерал, его можно передать вне круглых скобок:

```
lock (myLock) {  
    // Код, который необходимо выполнить  
}
```

Это делает подобные функции больше похожими на привычные управляющие конструкции. Кроме того, действует еще одна конвенция: если функциональный литерал имеет ровно один параметр, этот параметр можно не объявлять, и он автоматически получает имя `it` (а его тип выводится из контекста). Это облегчает использования функций, принимающих *предикаты*, таких как внешняя функция `filter` для класса `Collection`:

```
fun <T> Collection<T>.filter(accept : fun(t : T) : Boolean) : Collection<T>
```



Эта функция возвращает лишь те элементы исходной коллекции, на которых предикат `accept` возвращает `true`, например:

```
intList.filter {it % 2 == 0} // Возвращает только четные элементы исходного списка
```

### **Модули, компиляция и сборка**

Модуль как единица компиляции. Возможности оптимизации

Зависимости между модулями.

Сборочный скрипт как способ описания модулей

### **Заключение**

Наши планы, диалог с сообществом, читайте тут, пишите тут