JAVAWORLD

FEATURE

Why Kotlin? Eight features that could convince Java developers to switch

What would Java look like if someone designed it from scratch today? Probably a lot like Kotlin

By John I. Moore, Jr.

Professor of Mathematics and Computer Science, JavaWorld

SEP 18, 2019 12:08 PM PDT

Officially released in 2016, Kotlin has attracted a lot of attention in recent years, especially since Google announced its support for Kotlin as an alternative to Java on Android platforms. With the recently announced decision to make Kotlin the preferred language for Android, you may be wondering if it's time to start learning a new programming language. If that's the case, this article could help you decide.

Kotlin's release history

Kotlin was announced in 2011, but the first stable release, version 1.0, didn't appear until 2016. The language is free and open source, developed by JetBrains with Andrey Breslav serving as its lead language designer. Kotlin 1.3.40 was released in June 2019.

About Kotlin

Kotlin is a modern, statically-typed programming language that features both object-oriented and functional programming constructs. It targets several platforms, including the JVM, and is fully interoperable with Java. In many ways, Kotlin is what Java might look like if it were designed today. In this article I introduce eight features of Kotlin that I believe Java developers will be excited to discover.

1. Clean, compact syntax
2. Single type system (almost)

3. Null safety
4. Functions and functional programming
5. Data classes
6. Extensions
7. Operator overloading
8. Top-level objects and the Singleton pattern

Hello, World! Kotlin versus Java

Listing 1 shows the obligatory "Hello, world!" function written in Kotlin.

Listing 1. "Hello, world!" in Kotlin

```
fun main()
  {
    println("Hello, world!")
  }
```

As simple as it is, this example reveals key differences from Java.

1. `main` is a top-level function; that is, Kotlin functions do not need to be nested within a class.
2. There are no `public static` modifiers. While Kotlin has visibility modifiers, the default is `public` and can be omitted. Kotlin also doesn't support the `static` modifier, but it isn't needed in this case because `main` is a top-level function.
3. Since Kotlin 1.3, the array-of-strings parameter for `main` is not required and may be omitted if not used. If needed, it would be declared as `args : Array<String>`.
4. No return type is specified for the function. Where Java uses `void`, Kotlin uses `Unit`, and if the return type of a function is `Unit`, it may be omitted.
5. There are no semicolons in this function. In Kotlin, semicolons are optional, and therefore line breaks are significant.

That's an overview, but there's a lot more to learn about how Kotlin differs from Java and, in many cases, improves on it.

1. Cleaner, more compact syntax

Java is often criticized for being too verbose, but some verbosity can be your friend, especially if it makes the source code more understandable. The challenge in language design is to reduce verbosity while retaining clarity, and I think Kotlin goes a long way toward meeting this challenge.

As you saw in Listing 1, Kotlin does not require semicolons, and it allows omitting the return type for `Unit` functions. Let's consider a few other features that help make Kotlin a cleaner, more compact alternative to Java.

Type inference

In Kotlin you can declare a variable as `var x : Int = 5`, or you can use the shorter but just as clear version `var x = 5`. (While Java now supports `var` declarations, that feature did not appear until Java 10, long after the feature had appeared in Kotlin.)

Kotlin also has `val` declarations for read-only variables, which are analogous to Java variables that have been declared as `final`, meaning the variable cannot be reassigned. Listing 2 gives an example.

Listing 2. Read-only variables in Kotlin

```
val x = 5
...
x = 6   // ERROR: WILL NOT COMPILE
```

Properties versus fields

Where Java has fields, Kotlin has properties. Properties are declared and accessed in a manner similar to public fields in Java, but Kotlin provides default implementations of accessor/mutator functions for properties; that is, Kotlin provides `get()` functions for `val` properties and both `get()` and `set()` functions for `var` properties. Customized versions of `get()` and `set()` can be implemented when necessary.

Most properties in Kotlin will have backing fields, but it is possible to define a *computed property*, which is essentially a `get()` function without a backing field. For example, a class representing a person might have a property for `dateOfBirth` and a computed property for `age`.

Default versus explicit imports

Java implicitly imports classes defined in package `java.lang`, but all other classes must be explicitly imported. As a result, many Java source files start by importing collection classes from `java.util`, I/O classes from `java.io`, and so forth. By default, Kotlin implicitly imports `kotlin.*`, which is roughly analogous to Java importing `java.lang.*`, but Kotlin also imports `kotlin.io.*`, `kotlin.collections.*`, and classes from several other packages. Because of this, Kotlin source files normally require fewer explicit imports than Java source files, especially for classes that use collections and/or standard I/O.

No call to 'new' for constructors

In Kotlin, the keyword `new` is not needed to create a new object. To call a constructor, just use the class name with parentheses. The Java code

```
Student s = new Student(...);   // or var s = new Student(...);
```

could be written as follows in Kotlin:

```
var s = Student(...)
```

String templates

Strings can contain *template expressions*, which are expressions that are evaluated with results inserted into the string. A template expression starts with a dollar sign ($) and consists of either a simple name or an arbitrary expression in curly braces. String templates can shorten string expressions by reducing the need for explicit string concatenation. As an example, the following Java code

```
println("Name: " + name + ", Department: " + dept);
```

could be replaced by the shorter but equivalent Kotlin code.

```
println("Name: $name, Department: $dept")
```

Extends and implements

Java programmers know that a class can `extend` another class and `implement` one or more interfaces. In Kotlin, there is no syntactic difference between these two similar concepts; Kotlin uses a colon for both. For example, the Java code

```
public class Student extends Person implements Comparable<Student>
```

would be written more simply in Kotlin as follows:

```
class Student : Person, Comparable<Student>
```

No checked exceptions

Kotlin supports exceptions in a manner similar to Java with one big difference–Kotlin does not have checked exceptions. While they were well intentioned, Java's checked exceptions have been widely criticized. You can still `throw` and `catch` exceptions, but the Kotlin compiler does not force you to catch any of them.

Destructuring

Think of *destructuring* as a simple way of breaking up an object into its constituent parts. A destructuring declaration creates multiple variables at once. Listing 3 below provides a couple of examples. For the first example, assume that variable `student` is an instance of class `Student`, which is defined in Listing 12 below. The second example is taken directly from the Kotlin documentation.

Listing 3. Destructuring examples

```
val (_, lName, fName) = student
// extract first and last name from student object
// underscore means we don't need student.id

for ((key, value) in map)
  {
    // do something with the key and the value
  }
```

'if' statements and expressions

In Kotlin, `if` can be used for control flow as with Java, but it can also be used as an expression. Java's cryptic ternary operator (?:) is replaced by the clearer but somewhat longer `if` expression. For example, the Java code

```
double max = x >= y ? x : y
```

would be written in Kotlin as follows:

```
val max = if (x >= y) then x else y
```

Kotlin is slightly more verbose than Java in this instance, but the syntax is arguably more readable.

'when' replaces 'switch'

My least favorite control structure in C-like languages is the `switch` statement. Kotlin replaces the `switch` statement with a `when` statement. Listing 4 is taken straight from the Kotlin documentation. Notice that `break` statements are not required, and you can easily include ranges of values.

Listing 4. A 'when' statement in Kotlin

```
when (x)
  {
    in 1..10 -> print("x is in the range")
    in validNumbers -> print("x is valid")
    !in 10..20 -> print("x is outside the range")
    else -> print("none of the above")
  }
```

Try rewriting Listing 4 as a traditional C/Java `switch` statement, and you will get an idea of how much better off we are with Kotlin's `when` statement. Also, similar to `if`, `when` can be used as an expression. In that case, the value of the satisfied branch becomes the value of the overall expression.

Switch expressions in Java

Java 12 introduced switch expressions. Similar to Kotlin's `when`, Java's switch expressions do not require `break` statements, and they can be used as statements or expressions. See "Loop, switch, or take a break? Deciding and iterating with statements" for more about switch expressions in Java.

2. Single type system (almost)

Java has two separate type systems, primitive types and reference types (a.k.a., objects). There are many reasons why Java includes two separate type systems. Actually that's not true. As outlined in my article A case for keeping primitives in Java, there is really only one reason for primitive types--performance. Similar to Scala, Kotlin has only one type system, in that there is essentially no distinction between primitive types and reference types in Kotlin. Kotlin uses primitive types when possible but will use objects if necessary.

So why the caveat of "almost"? Because Kotlin also has specialized classes to represent arrays of primitive types without the autoboxing overhead: `IntArray`, `DoubleArray`, and so forth. On the JVM, `DoubleArray` is implemented as `double[]`. Does using `DoubleArray` really make a difference? Let's see.

Benchmark 1: Matrix multiplication

In making the case for Java primitives, I showed several benchmark results comparing Java primitives, Java wrapper classes, and similar code in other languages. One of the benchmarks was simple matrix multiplication. To compare Kotlin performance to Java, I

created two matrix multiplication implementations for Kotlin, one using `Array<DoubleArray>` and one using `Array<Array<Double>>`. Listing 5 shows the Kotlin implementation using `Array<DoubleArray>`.

Listing 5. Matrix multiplication in Kotlin

```
fun multiply(a : Array<DoubleArray>, b : Array<DoubleArray>) : Array<DoubleArray>
  {
    if (!checkArgs(a, b))
        throw Exception("Matrices are not compatible for multiplication")

    val nRows = a.size
    val nCols = b[0].size

    val result = Array(nRows, {_ -> DoubleArray(nCols, {_ -> 0.0})})

    for (rowNum in 0 until nRows)
      {
        for (colNum in 0 until nCols)
          {
            var sum = 0.0

            for (i in 0 until a[0].size)
                sum += a[rowNum][i]*b[i][colNum]

            result[rowNum][colNum] = sum
          }
      }

    return result
  }
```

Next, I compared the performance of the two Kotlin versions to that of Java with `double` and Java with `Double`, running all four benchmarks on my current laptop. Since there is a small amount of "noise" in running each benchmark, I ran all versions three times and averaged the results, which are summarized in Table 1.

Table 1. Runtime performance of matrix multiplication benchmark

Java (`double`)	**Java** (`Double`)	**Kotlin** (`DoubleArray`)	**Kotlin** (`Array<Double>`)
7.30	29.83	6.81	15.82

Timed results (in seconds)

I was somewhat surprised by these results, and I draw two takeaways. First, Kotlin performance using `DoubleArray` is clearly superior to Kotlin performance using `Array<Double>`, which is clearly superior to that of Java using the wrapper class `Double`. And second, Kotlin performance using `DoubleArray` is comparable to--and in this example slightly better than--Java performance using the primitive type `double`.

Clearly Kotlin has done a great job of optimizing away the need for separate type systems--with the exception of the need to use classes like `DoubleArray` instead of `Array<Double>`.

Benchmark 2: SciMark 2.0

My article on primitives also included a second, more scientific benchmark known as SciMark 2.0, which is a Java benchmark for scientific and numerical computing available from the National Institute of Standards and Technology (NIST). The SciMark benchmark measures performance of several computational routines and reports a composite score in approximate *Mflops* (millions of floating point operations per second). Thus, larger numbers are better for this benchmark.

With the help of IntelliJ IDEA, I converted the Java version of the SciMark benchmark to Kotlin. IntelliJ IDEA automatically converted `double[]` and `int[]` in Java to `DoubleArray` and `IntArray` in Kotlin. I then compared the Java version using primitives to the Kotlin version using `DoubleArray` and `IntArray`. As before, I ran both versions three times and averaged the results, which are summarized in Table 2. Once again the table shows roughly comparable results.

Table 2. Runtime performance of the SciMark benchmark

Java	Kotlin
1818.22	1815.78

Performance (in Mflops)