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

3. Null safety

One of the major design goals of Kotlin was to eliminate, or at least greatly reduce problems associated with null references. More than any other language I have used, Kotlin greatly reduces occurrences of the dreaded `NullPointerException`. The Kotlin type system distinguishes between references that can hold null values (nullable references) and those that can't, and the compiler will verify consistent usage of variables of each type. So, for example, in Kotlin, a variable of type `String` can never be null, but a variable of type `String?` (note `?` suffix) can. This idea is illustrated in Listing 6.

Listing 6. Null safety examples

```
var s1: String  = "abc"   // s1 is not nullable
var s2: String? = "abc"   // s2 is nullable
s1 = null    // compilation error
s1 = s2      // compilation error
s2 = s1      // o.k.
s2 = null    // o.k.

println(s1.length)   // will never throw NPE
println(s2.length)   // compilation error

...

fun printLength(s : String?)
  {
    if (s != null)
        println(s.length)   // o.k.
  }
```

Safe calls

If `s` is nullable, and if the compiler can't verify that `s` is not null at the point where `s` is dereferenced, then a call to `s.length` will be flagged as a compile-time error. However, a call to `s?.length` will return `s.length` if `s` is not null; otherwise it will return null. The type of `s?.length` is `Int?`, so after calling `s?.length`, the compiler checks that `Int?` is used safely. Since the function `println` can handle null values, a call to `println(s?.length)` is safe and will print either the length of the string or the word "null".

The Elvis operator

Frequently in programming we want to use a specific property of a nullable variable, so we check first to ensure that it isn't null. If it isn't, we use that property, but if it is, we want to use a different value in our code. Listing 7 illustrates this use case.

Listing 7. Example without the Elvis operator

```
val s : String?
...
val len : Int = if (s != null) s.length else -1
```

Kotlin has a "null coalescing" operator (?:), which is more popularly known as the "Elvis" operator, because if you turn your head slightly it resembles Elvis Presley's pompadour. The left side of the operator is an expression that can be null. If the value on the left of the operator is not null, then that value is used as the result of applying the operator. If the value on the left is null, then the value on the right of the operator is used. With the Elvis operator, the code in Listing 7 can be written more simply as shown in Listing 8.

Listing 8. Example with the Elvis operator

```
val s : String?
...
val len = s?.length ?: -1
```

The '!!' operator

In Kotlin, you can use the `!!` operator to tell the compiler to ignore checks for null safety. Consider the following code:

```
val len = s!!.length
```

Here you are saying to the compiler, "Look, I am confident that `s` is not null at this point, and I assume responsibility if it is. Do not flag this as a compile error." Of course execution of this code could still throw a `NullPointerException` at runtime if `s` is null.

4. Functions and functional programming

Functions in Kotlin can be top-level objects, meaning they do not need to be nested within a class. Moreover, Kotlin has other function-related features that support a cleaner, more compact syntax without loss of clarity. As an example, it's possible to omit the return type, curly braces, and return statement in single-expression functions. Therefore

```
fun double(x : Int) : Int
  {
    return 2*x
  }
```

could be replaced with the much shorter version below:

```
fun double(x : Int) = 2*x
```

Kotlin also allows default values for parameters, which can be used to eliminate some Java functions or constructors that simply call other functions or constructors with specific values for parameters. I often write constructors that call other constructors in the same class. For example, if I were to write my own `ArrayList` class in Java, I might provide a couple of constructors as follows:

```
public ArrayList(int initialCapacity)
  {
    ...
  }

public ArrayList()
  {
    this(10);
  }
```

In Kotlin this would be reduced to only one constructor:

```
constructor(initialCapacity : Int = 10)
  {
    ...
  }
```

Higher-order functions

Kotlin also supports higher-order functions, which means that functions can be passed as parameters to functions, returned as values from functions, or both. And, similar to Java, Kotlin supports lambda expressions and closures to avoid having to write complete function definitions. Moreover, if a lambda expression has only one parameter, then the parameter need not be declared, and the parameter will be implicitly declared using the name `it`. Additional features that support functional programming include optimized tail recursion and many common functional programming functions for lists. Listings 9 and 10 illustrate some of these features.

Listing 9. Functional programming in Kotlin (Example 1)

```
// assumes that students is a list of Student objects
val adultStudents = students.filter   { it.age >= 21}
                            .sortedBy { it.lastName }
```

Listing 10 below is from the article "My favorite examples of functional programming in Kotlin" by Marcin Moskala.

Listing 10. Functional programming in Kotlin (Example 2)

```
fun <T : Comparable<T>> List<T>.quickSort(): List<T> =
    if (size < 2) this
    else
      {
        val pivot = first()
        val (smaller, greater) = drop(1).partition {it <= pivot}
        smaller.quickSort() + pivot + greater.quickSort()
      }
```

In certain cases, Kotlin also supports inline functions, meaning that the compiler will replace a function invocation with the function body. Inlining a function can improve performance by eliminating call/return overhead, but it can also increase the size of your code, so use this technique cautiously. The place where inlining is most applicable is with lambda expressions, where inlining means that the compiler does not need to create a new function every time a lambda expression is created.

Iterable versus Sequence collections

For developers who are really into functional programming, Kotlin has a collection-like interface called `Sequence`, which is defined similarly to the `Iterable` interface but used in a manner similar to streams in Java or lists in Haskell.

Both `Iterable` and `Sequence` collections support a full range of functions from the functional programming paradigm, including `distinct`, `filter`, `map`, and `fold`. The essential difference between an `Iterable` and a `Sequence` collection is that sequences are "lazy," meaning that values aren't computed until needed. This makes sequences especially efficient for large collections. See "Kotlin sequences: An illustrated guide" for more information.

5. Data classes

Classes designed primarily to hold data are known as entity classes, business classes, or data classes. These types of classes are typically persisted to a database. In Java, these classes tend to have very few methods beyond constructors, `get()/set()` methods for fields, and standard methods `toString()`, `hashCode()`, `equals()` (and possibly `clone()`). While modern IDEs can save time by generating these standard methods for you, Kotlin provides a special kind of class, a data class, that doesn't require implementing these methods within the class.

You can grasp the benefits of data classes by comparing the Java code outlined in Listing 11 to the equivalent Kotlin code in Listing 12. Using methods generated by Eclipse, the full Java implementation consists of approximately 180 lines of uncommented source code while the Kotlin implementation consists of fewer than 10 lines of uncommented source code.

Listing 11. Data class example in Java

```
public class Student implements Cloneable
  {
    private String    studentId;
    private String    lastName;
    private String    firstName;
    private String    midInitial;
    private LocalDate dateOfBirth;
    private String    gender;
    private String    ethnicity;
    ...   // constructor with every field
    ...   // get()/set() methods for each field
    ...   // method toString()
    ...   // methods hashCode() and equals()
  }
```

Listing 12. Data class example in Kotlin

```
data class Student(var studentId   : String,
                   var lastName    : String,
                   var firstName   : String,
                   var midInitial  : String?,
                   var dateOfBirth : LocalDate,
                   var gender      : String,
                   var ethnicity   : String)
```

Records in Java

Java language architect Brian Goetz is working on a draft JDK Enhancement Proposal (JEP) that would add functionality similar to data classes to Java. The code example below (from the JEP) shows that the proposed new data classes, called records, are similar to Kotlin's data classes, with most of the standard methods derived automatically. Records have a name, a state description, and a body.

```
record Point(int x, int y) { }
```

See JEP Draft: Records and Sealed Types for more about the proposal to add records to Java.

6. Extensions

Extensions allow the addition of new functionality to a class without subclassing and without directly modifying the class's definition. Both extension functions and extension properties can be defined. An extension function is declared by prefixing its name with the name of the class being extended. Suppose, for example, we wanted to add a function to the `String` class. Listing 13 shows a simple example where we add a boolean function `isLong()`.

Listing 13. Extension function

```
fun String.isLong() = this.length > 30
...   // create string str
if (str.isLong()) ...
```

Similarly we can add an extension property to an existing class. For example, using the data class `Student` from above and without modifying the source code for the class, we can add extension properties `fullName` and `age` as illustrated in Listing 14. Note the required check for nullable property `middleInitial` in the implementation of the extension property `fullName`.

Listing 14. Extension properties

```
val Student.fullName : String
    get()
      {
        val buffer = StringBuffer(35)
        buffer.append(lastName)
              .append(", ")
              .append(firstName)
              .append(if (midInitial != null) " $midInitial." else "")

        return buffer.toString()
      }

val Student.age : Int
    get() = dateOfBirth.until(LocalDate.now()).getYears()

...   // create Student variable student

println(student.fullName)
println(student.age)
```

While these two properties are implemented as extension properties, they could also be implemented as computed properties within the definition of class `Student`, assuming access to the source code for `Student` is available. A computed property is essentially a `get()` method in Java without a backing field.

Note that calls to extension functions and references to extension properties are resolved statically by the compiler. This means the extension function being called is determined by the declared type of a variable and not by the type of the object the variable actually references at runtime. Polymorphism does not apply for extensions.

7. Operator overloading

In Kotlin, some functions can be called using predefined operators. Because I was a mathematician in my former life, this feature has strong personal appeal for me. Many mathematically related classes can be used in a more natural way when operators such as + and * are overloaded; examples include `Fraction` and `Matrix`. But even non-mathematical classes can benefit since the operator == can be used to call the `equals()` function, and brackets ([]) can replace calls to `get()` functions, as in `myList[n]` or `myMap["key"]`. C++ also allows operators to be overloaded, but personally I prefer the simpler approach used by Kotlin. Here is a partial list of common functions that can be called using operators.

Operator	Function name
+	plus()
*	times()
+=	plusAssign()
==	equals()
>	compareTo()
[]	get()
..	rangeTo()
in	contains()
++	inc()

The rules of operator overloading in Kotlin

It's important to understand a few points about overloading operators in Kotlin:

- First, functions that overload operators must be marked with the `operator` modifier. This is illustrated more clearly in the example shown in Listing 15 below. Note that the matrix multiplication function shown in Listing 5 can easily be converted to an operator function by adding the `operator` modifier and changing the name of the function from `multiply` to `times` (the name required by Kotlin for overloading the "*" operator).
- Second, when combined with Kotlin's single type system, you can use `==` almost exclusively without having to worry about when to use `==` and when to call the `equals()`. (Raise your hand if you have ever used `==` in a Java program when you should have called `equals()` method. That would be almost everyone, including me, and I strongly suspect that those with your hands down have not programmed extensively in Java.) Along these lines note that Kotlin uses the operator `===` for identity (a.k.a., referential equality). Also note that if variable `a` is null, then using `a == b` is safe since it translates to comparison for null using identity, as in `null === b`.
- Third, Kotlin *does the right thing* when the operator `++` calls `inc()`; this means the expression `++x` is compiled differently from `x++`, even though both expressions eventually call `inc()`. Similarly, Kotlin knows how to interpret the results from calling `compareTo()` when using relational operators such as `<=` and `>`.

Listing 15 provides an outline of a class that overloads operator functions, and Listing 16 illustrates the use of operators to call those functions. Note the use of braces {} in Listing 16 to include expressions within string templates.