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

Listing 15. Operator overloading for class fraction

```
class Fraction(numerator : Long, denominator : Long = 1) : Comparable<Fraction>
  {
    val numerator : Long
    val denominator : Long

    init
      {
        ...  // code to "normalize a fraction; e.g., 2/4 is normalized to 1/2
      }

    fun toDouble() = numerator.toDouble()/denominator.toDouble()

    operator fun plus(f: Fraction): Fraction
      {
        val numer = numerator*f.denominator + denominator*f.numerator
        val denom = denominator*f.denominator
        return Fraction(numer, denom)
      }

    operator fun minus(f: Fraction): Fraction
      {
        val numer = numerator*f.denominator - denominator*f.numerator
        val denom = denominator*f.denominator
        return Fraction(numer, denom)
      }

    operator fun times(f: Fraction)
        = Fraction(numerator*f.numerator, denominator*f.denominator)

    operator fun div(f: Fraction): Fraction
      {
        if (f.numerator == 0L)
            throw IllegalArgumentException("Divide by zero.")
        return Fraction(numerator*f.denominator, denominator*f.numerator)
      }

    operator fun inc() = Fraction(numerator + denominator, denominator)

    operator fun unaryMinus() = Fraction(-numerator, denominator)

    override fun toString() = "$numerator/$denominator"

    ...  // other functions such as compareTo(), hashCode(), and equals()
  }
```

Listing 16. Using class fraction

```
val f1 = Fraction(4, 10)   // normalized to 2/5
val f2 = Fraction(5)
val f3 = Fraction(2, 5)

println("f1 = $f1    f2 = $f2")   // f1 = 2/5    f2 = 5/1
println("f1.toDouble() = ${f1.toDouble()}")   // f1.toDouble() = 0.4
println("f2.toDouble() = ${f2.toDouble()}")   // f2.toDouble() = 5.0
println("f1 === f3 is ${f1 === f3}")   // f1 === f3 is false
println("f1 == f3 is ${f1 == f3}")     // f1 == f3 is true
println("f1 < f2 is ${f1 < f2}")       // f1 < f2 is true
println("f1.compareTo(f2) is ${f1.compareTo(f2)}")   // f1.compareTo(f2) is -1
println("-f1 = ${-f1}")           // -f1 = -2/5
println("f1 + f2 = ${f1 + f2}")   // f1 + f2 = 27/5
println("f1 - f2 = ${f1 - f2}")   // f1 - f2 = -23/5
println("f1 * f2 = ${f1*f2}")     // f1 * f2 = 2/1
println("f1 / f2 = ${f1/f2}")     // f1 / f2 = 2/25

var x : Fraction
val y = Fraction(1,2)
var z = Fraction(1)
x = y + z++
println("x = $x, z = $z")   // x = 3/2, z = 2/1

z = Fraction(1)
x = y + ++z
println("x = $x, z = $z")   // x = 5/2, z = 2/1
```

8. Top-level objects and the Singleton pattern

Some classes should have exactly one instance. These classes usually involve the central management of a resource such as a print spooler, a logger, a factory for a family of objects, an object that manages database connections, or an object that interacts with a physical device. Classes that can have exactly one instance with a global point of access are said to implement the Singleton pattern.

There are several ways to implement the Singleton pattern in Java, but if we're not concerned with thread safety, one common approach is to use a static field and a static `getInstance()` method, as shown in Listing 17 below. Note the use of lazy initialization in Listing 17, where the instance is not actually created until it is needed. One simple but often sufficient way to achieve thread safety is to make the `getInstance()` method synchronized.

Listing 17. Singleton pattern in Java

```
public class Singleton
  {
    // the one and only instance
    private static Singleton instance = null;

    ...  // other fields


    protected Singleton()
      {
        ...  // initialization code
      }

    public static Singleton getInstance()
      {
        if (instance == null)
            instance = new Singleton();
        return instance;
      }

    ...  // other methods
  }
```

Code using the Java singleton would look similar to the following:

```
Singleton.getInstance().someMethod();
```

In Kotlin, the Singleton pattern is implemented using a top-level object declaration. Note the use of "object" instead of "class" in the declaration in Listing 18.

Listing 18. Singleton pattern in Kotlin

```
object Singleton
  {
    ...  // properties and methods
  }
```

Code using the Kotlin singleton would look similar to the following:

```
Singleton.someMethod()
```

While Kotlin's approach is simple and straightforward, it is somewhat more complicated to implement a singleton in Kotlin if the constructor has parameters. See the end of this article for guidance on implementing parameterized singletons in Kotlin.

Additional features

Kotlin has more cool features than I could cover in one article. I will briefly mention two more that I think Java developers should know about.

Delegation

Think of *delegation* as an alternative to implementation inheritance. A class can implement an interface by delegating all of its public members to a specific object. Overriding a member of an interface implemented by delegation works as you might expect.

Coroutines

Think of *coroutines* as lightweight threads. Coroutines were considered experimental until Kotlin version 1.3. Like Java threads, Kotlin coroutines are not simple and can require considerable effort to master and to use effectively. It would take another whole article this size just to provide a complete overview of coroutines.

What's missing from Kotlin?

The answer to this question is ... not much. Still, there are a few Java features that I miss when working with Kotlin. They are not essential, and there are workarounds, but I find the Kotlin workarounds to be somewhat more awkward than their Java counterparts.

Coercion for numeric types

The first Java feature that I miss when I'm programming in Kotlin is type coercion (also known as implicit type conversion) for numeric types. In Java, if `n` has type `int` and `x` has type `double`, we can write `x = n` to assign the integer value to a double. The *coercion* from an integer type to a floating point type is well understood and has been present in most programming languages since Fortran. But in Kotlin, if `n` has type `Int` and `x` has type `Double`, we have to write `x = n.toDouble()`.

There is one exception to this restriction in that integer literals can be assigned to variables of the type `Byte` or `Short` provided that the literals are within the range of that type. This exception is permitted since the compiler can verify that the value is allowed for that type.

Kotlin's lack of type coercions for numeric types is a minor inconvenience, and you can read Kotlin's documentation to find out why the language requires explicit type conversion. Still, I occasionally miss simple numeric coercions when programming in Kotlin.

Static fields

The second Java feature I miss is the ability to declare `static` fields for a class. Evidently, Kotlin's language designers consider `static` fields to be less "object-oriented," but personally I was never bothered by the fact that some data values were stored with the class object rather than with each instance.

As an example, consider the `Fraction` class that I introduced in Listing 15. Let's say I want to include two constants representing the common values `0` and `1`. Listing 19 shows how these might be declared in Java.

Listing 19. Static fields in Java

```
public static final Fraction ZERO = new Fraction(0);
public static final Fraction ONE  = new Fraction(1);
```

Kotlin's solution is to create what it calls a *companion object*, which is defined within the `Fraction` class. Listing 20 illustrates how this would look in Kotlin. Again, this is only a minor inconvenience, but personally I like the Java version better.

Listing 20. A companion object in Kotlin

```
companion object   // defined within the Fraction class
  {
    val ZERO = Fraction(0)
    val ONE  = Fraction(1)
  }
```

It is frequently the case that `static` methods can be implemented as top-level functions in Java, but if you really need something roughly equivalent to a static method, it too can be implemented in a Kotlin companion object.

Conclusion: Is adopting Kotlin worth the pain?

Many programming languages were originally developed to overcome limitations or pain points of existing programming languages. Bjarne Stroustrup created C++ because he wanted a language with object-oriented capabilities and the speed of C, and James Gosling developed Java as a response to general dissatisfaction with C++. Newer languages like Kotlin, Scala, and Clojure have likewise emerged as alternatives to Java.

Will Google's promotion of Kotlin as the preferred language for Android be enough to push Kotlin to the forefront of developers' minds? I recently exchanged emails with noted author and Java expert Cay Horstmann, who had this to say about the adoption of programming languages:

> *Major language shifts occur when the new language offers something that is greater than the pain of switching. With C, the benefit was efficiency, portability, and a usable set of libraries. With Java, the benefit was garbage collection and an even more usable set of libraries.*
>
> *There are successful incremental changes, in particular from C to C++. But it was special because there was no pain. You could take your (ANSI) C code and it compiled as C++ code.*
>
> *I don't know if the incremental change from Java to Kotlin gives enough benefits to be worth the pain. Look at JavaScript, where there are saner alternatives, and the pain of switching prevents the switch from happening on a large scale.*

Java has been my preferred programming language for more than 20 years, and while it is a great language, it also has its share of detractors. Like all widely adopted programming languages, Java has evolved over time to address shortcomings and limitations and to improve programmer efficiency. But evolving an existing language requires strong consideration of compatibility with previous versions. Sometimes creating a new programming language is a better alternative. Is Kotlin better than Java? In many respects, the answer is "yes." Are the advantages of switching to Kotlin worth the pain? The jury is still out on that question, but clearly Kotlin deserves serious consideration for future software development projects, especially Android applications.

Learn more about Kotlin

- Visit the Kotlin homepage and Kotlin reference guide.
- Learn more about Kotlin's treatment of the Singleton pattern ("Kotlin singletons with argument," Christophe Beyls).
- Learn more about property delegation in Kotlin ("Kotlin explained: Property delegation," Samuel Urbanowicz).
- Kotlin Bootcamp for Programmers is an online learning course from Udacity.
- For your bookshelf, consider *Kotlin Programming: The Big Nerd Ranch Guide* (Josh Skeen and David Greenhalgh, Big Nerd Ranch, 2018) and *Kotlin in Action* (Dmitry Jemerov and Svetlana Isakova, Manning Publications, 2017).