Isaac Zinman

CS 162

Professor Hardekopf

31 January 2018

Learning Kotlin: A BOGOSudoku Journey

In this project I used Kotlin for the first time. Kotlin is a new language developed by JetBrains, the same folks behind the IntelliJ IDEA, and is primarily designed to target the JVM, especially Android devices (although it also compiles to JavaScript). Kotlinlang.org describes it as a "Statically typed programming language for modern multiplatform applications". What I found during my time learning Kotlin was an intriguing language that is syntactically more concise than Java and whose typing conventions attempt to solve some of the dreaded Java pitfalls like NullPointerExceptions, but suffers from the familiar problems of a young, little utilized language.

To begin, The Good. Coming to Kotlin as a Java developer is both reassuringly familiar and tantalizingly new. A few features that particularly stood out to me: first, the **in** keyword. This is a nice utility reminiscent of Python which can be used with **for** loops: **for(i in 1..10)**, as well as with range checks: **if (x in 1.5..7.5)**, and I found it an improvement over the C-style syntax I'm used to. Second, the way arrays are handled. In Kotlin arrays are objects, and although they must be initialized when declared which is annoying, the constructor encourages use of a lambda to initialize the values. I was able to declare and initialize a 2D array of random integers in a single line. It is certainly possible that this is doable in other languages but it was new to me, at least. Finally, one of the biggest advantages I see in Kotlin over Java is the

improved type system. One aspect of this is the addition of nullables - any time a variable reference might be **null** its type becomes nullable, denoted by the question mark (**?**). So when storing the result of an unsafe function, for example, one has to declare **var x : Double? = stringWhichMightNotContainInt.toInt()**, and **x** must now be null-checked before being passed to a function expecting a Double. I think this is an excellent way of forcing developers to write safer code, as it forced me to be very conscious of the possibility of null values as I used unsafe functions. Another aspect of the type system is **val** vs **var**, which is also an effective way of distinguishing constants from mutable variables and once again making you think about the code you're writing more carefully. Finally, functions have their own type Unit, which makes higher-order functions feel more natural.

The Bad. Unfortunately, Kotlin, as a new language, suffers from what one might call "growing pains." One *extremely* frustrating aspect of this project was my struggle to build and run my code, which proved difficult across IntelliJ IDEA, Eclipse, and even when using Gradle directly. I spent much more time on this aspect than actually coding, and it was very hard to find examples on Google, a challenge which carried over to other issues I had while coding. Eclipse seemed to be failing to re-compile .class files when their source files were updated; IntelliJ refused to configure its build system which eventually turned out to be because **main()** was within a **class main,** which according to the language spec should be valid, but I guess requires some kind of special build instructions; and even though I read a whole lot of documentation trying to learn Gradle to figure out how to write my own build script, I couldn't get that to work either. Another small thing is that Kotlin doesn't have the ternary operator **variable = (condition) ? val_if_true : val_if_false**, which is very strange since Kotlin code can

convert directly to Java code and Java has the ternary operator. Presumably as the language evolves and becomes more popular the IDEs' build tools will become smarter, the error messages will be easier to Google, and they will add stuff like the ternary operator.

The OOP. Object-oriented programming works largely similarly in Kotlin to Java. A few notable differences include the defaults of the language: classes are **final** by default and must be declared **open** to be heritable, and variables and functions within a class are scoped to **public** by default. I'm not sure if this is any improvement over Java. Kotlin has some interesting, concise syntax when creating classes: the primary constructor is declared in the same line as the class, and code to be run during instantiation can be placed in **init {}** blocks. I learned a bit about how this works as I tried using an **init {}** in my superclass to call an abstract function, thinking that this would cause subclasses to call their overridden version of the function, but that didn't work so I had to call it explicitly in the subclass constructor. Maybe this just has to do with the order of code execution as objects are constructed, but it seems like it suggests that Kotlin uses compile-time function binding rather than runtime (not sure if this is consistent with Java or not). However, with the limited amount of polymorphism I involved, Kotlin behaved basically the same as Java as far as inheritance goes – the simple inheritance hierarchy of Sudoku -> SudokuRandom -> SudokuValidated didn't involve any particularly distinct features.

In conclusion, Kotlin was interesting to learn and has some promising aspects, but the huge time investment into the build process ultimately made the experience a frustrating one. I will be interested to see how Kotlin evolves as it matures as a language.