Oakland University

Kotlin Research

by

Blair Sibu

in the

Computer Science Department

November 2017

Efficiency

Kotlin is a statically typed programming language. This means that the type of variable is known at compile time. However, even though it uses Java libraries to compile it is different in how it specifies its variable type. In Java each variable type must be specified, but in Kotlin it has a way to deduce the type of the variable.

JIT Compiler

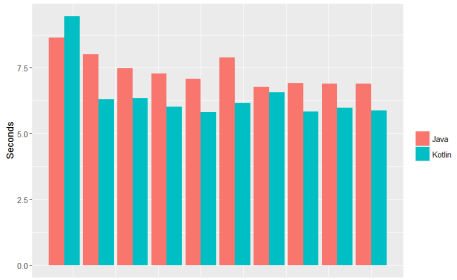
Since Kotlin relies heavily on Java libraries it also uses its Just-In-Time (JIT) compiler. The JIT compiler improves performance at runtime and converts the code into Java byte code contained in a .class file. On start-up thousands of methods need to be called which slows down initial compilation time. Even though it's calling thousands of methods, JIT keeps a strict track of how much memory it is using to avoid an overflow. Once it uses up its maximum cache the JIT compiler stops compiling methods.

Type Casting

Unlike Java, Kotlin can do type casting automatically on compilation. This would be a problem since it could hide some type compatibility errors, but Kotlin only automatically casts something when there is no other possible type to cast a variable to. One thing that cannot be done with Kotlin is assign primitive type to more complex types. In this case, they do not automatically type cast upon compilation. For example,
$val = x : Int = 20$, $val = y : Long = x$ will not compile. We must use a toLong() dot operator to x to explicitly convert it to a long type instead.

Compiler Time

When looking at Java v. Kotlin for Android Development it is only slower on clean builds of an app. On average Kotlin compiles slightly faster with incremental compilation.



incremental builds with 1 core file changed

Simplicity

For the readability and writability of Kotlin it is much easier than Java. The declaring of variables almost looks like javascript, and to declare and object the 'new' keyword isn't needed. Kotlin takes all of the code that can be written in Java and condenses it down in an almost shorthand-like form.

Writability

As stated earlier Kotlin takes a lot less code than Java does. For instance in Java to code a Plain Old Java Object (POJO) with getters, setters, equals(), hascode(), toString(), and copy it takes more than twenty lines of code. With Kotlin we can simply make the POJO in one line: `data class Person(var name: String, var age: Int)`. With the person class in place a person can be created like so: `val jack = Person(name = "Jack", age = 20)`. There is no need to use the 'new' keyword to create a POJO. Using the Java classes also gives it an advantage of being able to use all the things that Java can use such as maps and arraylists, so it brings the familiarity of Java in being able to write it easily. There are some more extreme sides to Kotlin. In Kotlin you cannot override a class or function unless it's open. Similarly you cannot inherit anything from another class unless it is open. A Good point brought up by Robert C. Martin is that Kotlin is less flexible because its a more statically typed language, even more so than Java, and it puts error checking that would normally be done by the programmer onto the language itself. Although Kotlin manages to compress a lot of unnecessary lines, as stated above, making it a lot easier to write it has some other unfavorable characteristics.

Readability

Reading Kotlin code is a little bit trickier than writing it. From the code above, `data class Person(var name: String, var age: int)` it is not clear that this is creating a POJO unless that information is known beforehand. Also many languages use the '?' ternary operator, and so does Kotlin, but it also uses it for its Null Safety feature. So depending on the context it could become a little confusing. There are some things that bring a little familiarity to a programmer, for example to declare variables the 'var' keyword is used. This is the same exact way local variables are created in C# and Javascript. In order to make constants easier to read they have allowed underscores to be used, for example if we have a credit card number it could be declared as:
$val\ creditCardNumber = 1234\_5678\_9012\_3456L$ where L indicates that it is of type long and val is the equivalent of 'final' in Java. The creators of Kotlin have also taken a page out of python's book by using the keyword 'in.' This makes the foreach loop more readable: `for(c in arr)`. Despite the few things that Kotlin does that make it confusing at times it reads a lot like other languages would. It takes on a lot of characteristics that Javascript, Java, and Python have. In doing so it is easier to read than Java.

Orthogonality

Kotlin has a relatively small set of primitives; they are the ones that we would expect from Java, integers, booleans, characters, floats, etc. It also uses the same kinds of data structures as Java such as the Array class. The Array class can be used as expected, that is it can be used with all of the primitive types like so: `Array<TYPE>`. Maps, Lists, etc can be used in the same way. This leaves us with fairly regular amount of Orthogonality.

Since Kotlin is so concisely typed there is only one way in which we can declare these structures. [need to write more here]

Definiteness

Syntax and Semantics

If we compare the syntax of Kotlin to Java almost everything is entirely new. Instead of declaring a variable with its data type, like a lot of the strongly typed languages, Kotlin starts the declaration with either val or var. Val means that the value the variable takes on is immutable while var is mutable. One plain example in how things are declared is the declaration of primitive arrays, that is arrays that are not declared using the Array class: *val x: IntArray = intArrayOf(1, 2, 3)*; which means the value x, which is immutable, is an integer array containing the values 1, 2, and 3. Values from this array can be called as expected, x[0]. Also all of the types (int, boolean, double, etc) are declared after the variable name. However, as stated previously there are times where we do not need to declare the type of the variable because the compiler can infer.

Kotlin also uses the '?' operator to check if something will output a null, or simply to box variables into say the java.lang.Integer class: *val bob?.department* or *val a: Int? = 1* respectively. The first one checks to see if bob is in a department, if he is it will return the value of the department otherwise it will return null. For the second one it is checking to see if the following number can safely be boxed into the java.lang.Integer class. Another useful syntax is when an argument has to call upon a range of numbers, *if(c in 0..9)*. When there's an argument that is searching through a range of numbers it simply means that if the c is in the range from zero to nine, then execute. The range can be used in any conditional statement. '!in' could also be used which is the negation of the 'in.'

For many other structures, if-else, while, for, etc they are all written the same as in Java. Looking back at the example of the foreach loop *for(c in str)*, this means precisely that for each element c, that is for each character, in the string variable str. So the loop will loop as long as a character remains in the string str. Since there are many primitive types there are also many ways to convert them, for example if an argument needed a long but instead the variable was an integer it could be changed with the following: *x.toLong()* assuming that x is an integer. This can also be done with strings to characters or a numerical to a character. A function can be define with the keyword 'fun.' It is much like a method in Java and it takes input quite the same way, for example *fun printer(str: String)*. This function is named printer and take a string input. For anything that is syntactically the same in Kotlin as it is in Java, the semantics are also the same.

Reliability

Kotlin has just as much reliability as Java does and more. With Kotlin there's the Null safety feature which protects us against the null pointer value exception. It also uses the try catch blocks as well as the throwable Java class.

Null Safety

Kotlin tries to get rid of Java's NullPointerException error. To do this Kotlin explicitly tells the system which references can hold null. It does this with the '?' ternary operator. For example, *var b: String? = "abc"*, if at any point the variable b becomes null it's ok because it has been allowed to be nullable. However, this does not mean that error checking still doesn't have to be done. If b does happen to become null and later on in the program it's using b.length, then the program must first check if b is null before performing the length command. This can be done like so, *if(b != null) b.length)*. Although instead of checking it to be null there's another operator, '!!'. This operator says that the variable has to strictly be non-null to perform the operation, otherwise it returns a NullPointerException.

Exceptions