Olivier Dorvelus

9/19/17

Kotlin

Kotlin is a fairly new programming language that was announced in 2011 by JetBrains. JetBrains wanted a modern feel like language to run in the JVM similar to Scala but without the slow compile time that Scala was known for. JetBrains IDEs are written primarily in Java so JetBrains wanted a more expressive language. The people of JetBrains also did not want to sacrifice giving up Java on their existing codebase to they planned for Kotlin to be 100 percent interoperability with Java. This way, programmers who wanted to integrate Kotlin in their projects could do so without starting from scratch. 100 percent interoperability also allowed for the compile time on existing programs to remain roughly the same. Java’s JDK 6 is the first JDK that Kotlin was able to run on. The lead developer of Kotlin, Andrey Breslav wanted companies with Java codebases to slowly migrate to Kotlin. Since Kotlin is so closely related to Java, there is a good learning curve when it comes to Kotlin for Java developers or even programmers that use the object orientated paradigm. Kotlin is surging in popularity due to its expressive nature and basically being a shorthand java reducing java’s boilerplate code. An interesting factor of Kotlin is that developers decided to let the programmer decided what paradigm they want to choose. Kotlin supports object orientated similar to Java but also supports functional programming. Kotlin was designed in order to be a better, more modern Java while still maintaining strong ties with Java in order to not alienate programmers. Since Kotlin runs in the JVM, it is capable of running anything Java can but Kotlin is made primary for server side applications. Kotlin can run on any Java web application and before official support for android, there was plugin support in order to use Kotlin in android studio. Finally, a modern, more concise language can be used for developing android applications while maintaining compile time and code familiarity for crossover or new development.

The structure of Kotlin is a very optional kind of structure. For example, being able to implement either the object oriented or function paradigm. Kotlin is a language that allows the programmer to choose from many different options to execute the same functionality. For example, Kotlin at its core is a statically typed language but allows for dynamically typed implementation.

var first\_name = "Olivier"  
var last\_name:String = "Dorvelus"  
var num1:Int = 3;  
var num2 = 4;

These are all valid statements in Kotlin, you can define the variable both implicitly and explicitly. The **var** keyword is a must when assigning a variable. After the variable name you can put an : symbol and the variable type after or you can assign the value right after the variable name. Another concept to note is that semicolons at the end of each statement is completely optional. This is practically the best of both worlds for legacy programmers versus modern programmers. Just like many other languages you can reassign variables but you can declare immutable variables using the **val** keyword instead of var.

var city:String = 400  
num2 = 3  
num2 = "I'm a String"

var num3

var num4:Int

The first line gives an error because a string is declared explicability but was assigned a different data type. The second line is fine but the third line will result into an error because Kotlin at its core is still a statically typed language. You can’t redefine a data type once it is assigned. The fourth line is also incorrect, a type must be declared if the variable is not assigned a value such as line 5 Remember, Kotlin is compiled down to Java bytecode. Kotlin has a main function which is where the program looks at to execute such as Java, C++, and C#.

fun main(args : Array<String>) {

// body

}

Whitespace is ignored in Kotlin and keeps the curly bracket syntax that usually denotes the body of a class, function, or loop. Print or println handles outputting and allows the traditional concatenation to combine strings with variables or string substitution. Kotlin has a very smooth programmer friendly way of implementing variables with outputs.

*println*("${x+y} $name")

A $ sign followed up by the variable name in a string, if multiple variables are needed for substitution then they must be encased in a curly bracket after the $ symbol.

Kotlin includes the control structures most programmers are familiar with such as if else statements and different types of loops such as for, while, when, and more. An if statement must take a Boolean as a parameter to determine whether that statement is true or false. Depending on that condition, then a certain block of code is executed instead of other blocks. Conditional statements are also equipped with the else if and else statement. Else if also evaluates an boolean expression and runs that block of code if that condition is true. The block of code that is executed is the first true expression, top to bottom the compiler runs into. The program below is only going to print out Hello World despite both conditions being true. Lastly, there is an else statement. The block of code in an else statement will run if none of the conditions are true.

if(2 == 2) {  
 *println*("Hello World")  
}  
  
else if(2 == 2) {  
 *println*("Hello Mars")  
}

You are also allowed to set variables equal to the result of a conditional statement in Kotlin.

result = if(num1 > num2) {  
 num1  
}  
else {  
 num2  
}

The variable result is going to be equal to either num1 or num2 depending on the condition. Lastly for conditionals, Kotlin replaces the tradition switch case implementation using a when implementation. When is a keyword that takes in a value and evaluates a list of candidates that and will execute the block that is equivalent to that value. When expression must contain an else incase none of the values match. Kotlin also allows a variable to equal the result of a when expression.

var num = 2  
when(num) {  
 1 -> *println*("One")  
 2 -> *println*("Two")  
 3 -> *println*("Three")  
 else -> *println*("Give a proper input")  
}

Moving on to iterators, Kotlin handles for loops very differently from Java. Kotlin takes a more modern route similar to dynamically typed languages such as Python and Ruby. Kotlin for loops are more like foreach loops. A for loop can be used with any object that has a built in iterator such as arrays, list, and hashes. The syntax below will loop through every item contained in that data structure.

for(item in Array) {  
 // do ...  
}

For loops also allow for range, which can be used to replace the traditional for loop.

for(i in 1..10) {  
 *println*(i)  
}

This will print out numbers 1-10, inclusive. Unfournately, the option of using the for loop syntax in how Java uses it is not available in Kotlin. While and do while loops are implemented exactly the same in how they are in Java. A while loop will keep executing a block of code as long as the Boolean expression in it is true. A do while loop differs because it will at least execute a block of code once even if the expression is false. The program below will print numbers one through ten. The do while loop is still going to output one despite the condition being false because a while loop executes the do before evaluating the condition. For the most part, the developers of Kotlin kept control structures similar to C based languages.

var counter = 1  
while(counter < 11) {  
 *println*(counter)  
 counter++  
}

counter = 1

do {  
 *println*(counter)  
}  
while(counter < 0)

­­­ Kotlin contains many different type of data structures that helps the programmer builds fast cohesive programs while reducing boilerplate code. Since Kotlin is 100% interoperable with Java, you can import any Java library into a Kotlin file. Kotlin represents numbers very closely with Java but not exactly the same. There are no implicit widening conversions for numbers, re are no implicit widening conversions for numbers. Integers can range from 8-64 bits, starting with byte all the way to double. Kotlin also allow number variables to be more readable by using number literals.

var phonenumber = 347\_840\_8556  
 *println*(phonenumber)

The program still outputs 3478408556 since there is no string formatting but it allows large values become more readable. Kotlin provides methods that will try to convert a datatype to another datatype such as toInt(). Usually with these operations, Kotlin provides a try catch exception statement. Try catch exception statements is for trying to run a block of code that might not always work. Below, the program will successfully convert the string since it’s a number but it cannot convert something like “Hello World” to an integer and therefore will run the exception error block of code.

var str : String = "45"  
 num = str.*toInt*()  
}  
catch(e : Exception) {  
 *println*("Give Proper Input")  
}

In Kotlin, shorter byte number representations are no longer subtypes of larger ones. Implicit conversions are handled by Kotlin, for example adding an integer and a double, the result will automatically be a double. Boolean expressions are something that can be evaluated as false or true. Boolean operators in Kotlin are represented the same in many C based languages with ||(or), &&(and), less than, equal to, and more. A character is represented with single quotes ‘c’. A string is an immutable array of characters in Kotlin. Arrays in Kotlin are represented by the Array Class that provide many different functions such as size, retrieving an element and more. Take note that many data types come with convenient operations that programmers use. Kotlin allows different data types to be in arrays like a dynamic language so a programmer is allowed to have an integer along with a string as an element. Kotlin still allows an array to only be one type with a different kind of declaration. Lastly, Kotlin handles the null data type in an interesting way. Kotlin rids of the null data type unless explicitly declared by the user to avoid one of the most common Java errors, the null pointer exception.

var name: String**?** = null

var nums = *intArrayOf*(2, 3,4,5, 6)  
var random = *arrayOf*("Olivier", 3, true, 10)

The type arrays aren’t inherited from the array class but perform the same operations. Elements in arrays are still accessed to square bracketing starting from zero. Kotlin comes with its own set of built in data types that many programmers are familiar such as hashes, trees, and list with convenient operations.

Kotlin also takes a different approach to user defined or composite data types. This is where Kotlin really reduces boilerplate code to familiar Java programmers. Kotlin automatically takes care of getters and setters, so there is no need to worry about information hiding. Once a class is initialized with instance variables and methods, Kotlin takes care of security in the background.

class Pokemon(var name:String, var level:Int) {  
   
}

Default constructors are decided in the actual class definition. Secondary constructors are used by using the **constructor** keyword. The parameters of secondary constructors must contain that of the default plus any extra parameters the user wants. A cool feature of Kotlin is the **data** keyword before defining a class. A data class automatically provide functionality of copying, toString, and a hash code for comparisons. In Java, these were operations a user had to keep defining for each class they made. All classes in Kotlin are automatically final, meaning no inheritance by default. Use open keyword to change it. Kotlin does not support multiple inheritance because Java doesn’t. Kotlin classes also contain the **init** keyword which is a function that runs every time a class instance is initiated. Default values for constructors can also be provided by setting the variable equal to a value in the header, this is the same for functions as well.

open class Human(age : Int = 3) {  
 init {  
 *println*("in human $age")  
 }  
 open fun think() {  
 *println*("Real Thinking")  
 }  
}  
  
class Alien(age: Int) : Human(20) {  
 init {  
 *println*("in alien")  
 }  
 override fun think() {  
 *println*("Virtual Thinking")

//inheritance

class Human (var n : String) {  
 var age : Int = 0  
 var name : String = n  
  
 constructor(age: Int, name : String) : this(name) {  
 this.age = age  
 }

Kotlin also supports abstract classes using the **abstract** keyword before a class definition. In order to achieve multiple inheritance, Kotlin implements interfaces. Usually interfaces are not supposed to define functions but Kotlin allows it. Kotlin also provides **companion object,** this is a way to call a function in a class without having an instance for that class.

interface A {  
 fun show()  
}  
  
interface B {  
 fun display()  
}  
class C : A,B {  
 override fun show() {  
 *println*("in show")  
 }  
  
 override fun display() {  
 *println*("in display")  
 }  
}

class A {  
 companion object {  
  
 @JvmStatic  
 fun show() {  
 *println*("Hello")  
 }  
 }  
}

Kotlin handles its name structures similar to how C languages handles it. Many variables are local to its certain block of code. This applies to if statements, loops, functions and classes. If a variable is initialized in a block of code, then its scope is in that block of code.

while(true) {  
 var counter: Int = 0  
 while (counter < 5) {  
 *println*(counter)  
 counter++  
 }  
 return  
}

Since counter is initialized in the first while block, it cannot be accessed outside that block. This goes with anything that can be encased with curly braces such as functions, classes, and control statements. One way to make a global variable in Kotlin is to simply declare it outside of anything that contains a brace, not even the main function. When you reference the variable, it will look toward that global variable in the file.

Kotlin primarily keeps the grammatical conventions of many C languages with the options of making the language more programmer friendly. For example, Kotlin allows the programmer to put semi colons at the end of each statement like most C languages but this is completely optional. Kotlin keeps the familiar curly brackets that Java uses. Curly braces are good for keeping track of what block of code belongs to what class, function, loop, if statement and more. Kotlin also shorthand a lot of reserved keywords that are in Java such as function being replaced with just fun. This supports the object orientation paradigm because a programmer can clearly see what belongs to a class, a function since it will be enclosed in a curly bracket. This also helps to decide the difference between local and global variables.

Kotlin can very well be the future of android programming and take over programs that are currently running in the JVM. Kotlin goal is to reduce boilerplate code, and removing many of the errors Java programmers run into such as null pointers and casting. Kotlin also aim to be just as fast as the statically typed languages. Kotlin is the future of android programming, Google has officially implemented Kotlin support in android studio. Several known companies such as Pinterest, Square, and Basecamp are converting their codebase into Kotlin. It is well known the less lines of code a program has, the less likely the program will encounter bugs. Kotlin succeeds very well at its goal and programming in it has been a true pleasure whether creating trivial programs or building an app with android studio. As a big fan of Java, Kotlin is basically a shorthand java that reduces repetitive code and errors. Kotlin is simply more fresh and can easily creep into the top ten programming languages. It’s great for newcomers and seasoned Java developers.