Introduction to Kotlin

Kotlin is a new language that essentially makes Java look like the Swift programming language. Kotlin runs on the JVM, so you can access all of the Java class libraries using the Kotlin syntax. One big difference is that Kotlin does not require semi-colons to mark the end of a line of code.

Introduction

Kotlin is currently supported by IntelliJ. Download the free community version of the lDE here:

<https://www.jetbrains.com/idea/download/>

When installing, ensure to include the Kotlin plugin.

1. Variable Declaration. This section describes how to declare variables in Kotlin.

The Kotlin compiler uses type inferencing to declare variables. You declare a variable using the **var** keyword and set it equal to an initial value. For example:

*var index = 0*

*var message = “Hello world”*

declares a variable called index of type integer and sets it initially to 0. It also declares a variable called message of type String and sets it to Hello world.

* 1. Constants: Variables have an initial value, but they can hold other values at any time in the program. Constants are set to an initial value and can never change. This is done using the **val** keyword. The following code is similar to the example above, but index and message can never store any other values:

*val index = 0*

*val message = “Hello world”*

* 1. Variables without initial values: Variables can be declared without initial values. The variable types are specified using the class name. The variables use the lateinit keyword to say that they will be given values later:

*lateinit var index : Integer*

*lateinit var message : String*

You cannot use lateinit with val to declare constants. Constant values must be provided at declaration time.

* 1. Null checks: Traditionally, variables that could be null must be checked before accessing members or calling functions:

*if(message != null)*

*index = message.indexOf(“Hello”);*

Kotlin uses the ? operator to shorten null checks:

*index = message?.indexOf(“Hello”)*

Kotlin also has an non-null assertion operator (!!). If the object is null, it will throw an assert failed exception:

*index = message!!.indexOf(“Hello”)*

Any line following the non-null assertion no longer requires null checks:

*message.toUpperCase()*

*message.length()*

* 1. Static variables: Variables that belong to the class, instead of to each object, are declared using a ***companion object{ }*** declaration. Everything in the braces are static variables. For example in java:

*static final String TAG\_NAME = “NAME”;*

*static final String TAG\_AGE = “AGE”;*

*static final String TAG\_ADDRESS = “ADDRESS”;*

would be declared in Kotlin:

*companion object{*

*val TAG\_NAME = “NAME”*

*val TAG\_AGE = “AGE”*

*val TAG\_ADDRESS = “ADDRESS”*

*}*

* 1. String replacement: Kotlin uses the $ to replace variable names with their value. The string

“*My* ***$TAG\_NAME*** *is Eric*” would evaluate to: “*My* ***NAME*** *is Eric*”

* 1. Casting: Kotlin uses casting functions: ***toXXX(*)**. Calling toString() returns the String representation of the object. Other functions are toLong(), toInt(), toDouble(), toBoolean(), etc.
  2. Properties: Kotlin supports code validation for getter and setter functions. The syntax follows C# syntax:

var name: Type

get() { return … }

set(value) { }

1. Function declarations. Functions are declared using the fun keyword followed by the UML syntax for functions:

fun functionName( parameter1: Type1, parameter2: Type2) : returnType

The first part of the declaration is the function name. The parameters are declared in the parentheses using the parameter name followed by the parameter’s type. The return type comes after the closing parenthesis. If the function does not return anything then the return type is not required. For example:

*fun getValueForNameFromFile( name: String, file:String) : String*

is a function called **getValueForNameFromFile**, which has 2 String parameters: **name**, **file**, and returns a **String**.

* 1. Default parameters. Kotlin supports default parameters in functions. The default value is specified by: ***= value***. The parameters must be specified from right to left in the function signature. For example:

*fun getValueForNameFromFile( name: String, file:String = “file.txt” ) : String*

* 1. Static functions. Static functions are declared like static variables, using the companion object{} syntax:

*companion object{*

*fun main(args:Array<String>)*

*}*

You should notice that this is the entry point to any Java program:

public static void main(String [] args)

However it is also requires @JvmStatic annotation in Kotlin:

*companion object{*

*@JvmStatic fun main(args:Array<String>)*

*}*

\*Note Java and Kotlin are inter-mixable within a project, however to access a static Kotlin property from java, you have to use the companion object scope:

In Java:

*AKotlinClass.Companion.main( new String[] {} );*

The @JvmStatic annotation lets Java access the function directly without the Companion scope specifier:

In Java:

*AKotlinClass.main( new String[] {} );*

* 1. Lambda functions Functions are wrapped in curly braces { }. The function parameters are the first part of the braces, followed by the -> operator. Everything after is the function body. For example:

***{ e, f -> foo() }***

***e*** and ***f*** are the function parameters, and ***foo()*** is the function body. If the parameters are not used in the function body, then they can be replaced with \_ :

***{ \_, \_ -> foo() }***

For passing lambda functions, they are written like Java:

***setOnClickListener( { e -> println(“Hello world”) } )***

If the lambda function is the last parameter in the function’s parameter list, then you can omit the parenthses ( ):

***setOnClickListener { e -> println(“Hello world) }***

* 1. Anonymous functions Anonymous functions are declared like normal functions, only the function name is omitted:

***fun(x: Int, y: Int): Int {***

***return x + y***

***}***

Here is an example of storing an anonymous function in a variable **g**:

***var g = fun (c:Int):String = c.toString()***

and to invoke it:

***g(4)*** //this returns the string “4”

1. Access Modifiers: Kotlin uses ***public***, ***private***, ***protected*** as in Java. The default behavior is ***public*** if no modifier is used. There is a 4th modifier, ***internal***. Every class in the same package can view those variables and functions.
2. Control structures. Kotlin uses the same syntax for if, while, do/while loops as Java.
   1. Switch: Kotlin has a different syntax for switch/case however. The syntax is:

when(value){

value1 -> { } // result code goes in brackets.

value2 -> { }

value3, value4 -> { } // this shows two values have the same code

else -> { } // this is similar to default: case.

}

The syntax is similar to lambda functions when the condition matches the value.

* 1. For loops: For loops use a range operator (begin .. end) to determine the loop. An example:

***for( i in 0..9)***

Counting downwards uses the *downTo* keyword:

***for( i in 9 downTo 0)***

Iterating by more than 1 uses the *step* keyword:

***for(i in 0 .. 9 step 2)***

* 1. For loops for arrays: For loops can also iterate over arrays or collections:

An example:

***for( obj in Array)***

1. Class definitions. Classes are defined using the ***class*** keyword. Inheritance is specified using the **:** operator. Classes can only inherit from one parent, but can implement several interfaces by listing them using commas:

***class MyClassName : parent , interface1, interface2 {***

***}***

Interfaces are declared as in Java:

***interface MyInterfaceName{ }***

* 1. Constructors: Kotlin has the notion of a primary constructor, and is written in the class definition:

*class MyClassName(* ***var age:Int, var name:String****) : parent { }*

This allows to declare member variables and a constructor at the same time. The above line is similar to the following in Java:

*class MyClassName extends parent{*

***public int age;***

***public String name;***

***public MyClassName(int \_age, String \_name)***

***{***

***age = \_age;***

***name = \_name;***

***}***

*}*

The drawback of the primary constructor in the class definition is that there is way to run extra code. To fix this, Kotlin allows init{ } blocks:

*init{*

*if(name == null)*

*name = “”*

*}*

Kotlin allows other secondary constructors using the ***constructor*** keyword, but must call the primary constructor using the ***this*** keyword:

constructor(\_age: Int) : this(age, “”)

{

age = \_age

}

constructor( \_name:String ) : this( 0, \_name)

{

}

1. Operator Overloading. Kotlin supports operator overloading. You can overload unary operators (+, -, !) by implementing the functions unaryPlus(), unaryMinus(), and not(). The Kotlin compiler translates the operators to the spelled out version of the function. For example, for an object **a**:
   * 1. +a is translated to: ***fun unaryPlus( ){ },***
     2. -a is translated to: ***fun unaryMinus() { }***
     3. !a is translated to: ***fun not() { }***
     4. a++ is translated to: ***fun inc() { }***
     5. a-- is translated to: ***fun dec() { }***
2. Binary operators:
   1. a + b is translated to: a.plus(b)
   2. a - b is translated to: a.minus(b)
   3. a \* b is translated to: a.times(b)
   4. a / b is translated to: a.div(b)
   5. a % b is translated to: a.rem(b)
   6. a..b is translated to: a.rangeTo(b)
   7. a in b is translated to: b.contains(a)
   8. a !in b is translated to: !b.contains(a)
   9. a[i] is translated to: a.get(i)
   10. a[i, j] is translated to: a.get(i, j)
   11. a[i\_1, ..., i\_n] is translated to: a.get(i\_1, ..., i\_n)
   12. a[i] = b is translated to: a.set(i, b)
   13. a[i, j] = b is translated to: a.set(i, j, b)
   14. a[i\_1, ..., i\_n] = b is translated to: a.set(i\_1, ..., i\_n, b)
   15. a() is translated to: a.invoke()
   16. a(i) is translated to: a.invoke(i)
   17. a(i, j) is translated to: a.invoke(i, j)
   18. a(i\_1, ..., i\_n) is translated to: a.invoke(i\_1, ..., i\_n)
   19. a += b is translated to: a.plusAssign(b)
   20. a -= b is translated to: a.minusAssign(b)
   21. a \*= b is translated to: a.timesAssign(b)
   22. a /= b is translated to: a.divAssign(b)
   23. a %= b is translated to: a.remAssign(b)
   24. a == b is translated to: a?.equals(b) ?: (b === null)
   25. a != b is translated to: !(a?.equals(b) ?: (b === null))
   26. a > b is translated to: a.compareTo(b) > 0
   27. a < b is translated to: a.compareTo(b) < 0
   28. a >= b is translated to: a.compareTo(b) >= 0
   29. a <= b is translated to: a.compareTo(b) <= 0

More Info: <https://kotlinlang.org/docs/reference/operator-overloading.html>

1. Threads. In Kotlin, threads are called coroutines. They are launched from an object called GlobalScope:

*GlobalScope.launch{*

*//thread logic here*

*}*

To wait for a thread’s completion:

*val job = GlobalScope.launch {*

*delay(1000L)*

*println("World!")*

*}*

*println("Hello,")*

*job.join() // wait until child coroutine completes*

1. Destructure Declarations. You can break an object down to its component variables based on the order of declaration in the constructor. For example, this declares a class with 3 variables:

***class Foo(var a:Int, var b: String, var c:Double)***

You can create an instance using the constructor:

***var obj = Foo(3, “Hello”, 7.5)***

To get the variables of the object, you can “destructure” the object:

***val (newA, newB, newC) = obj***

In this case, **newA = 3, newB = “Hello”,** and **newC = 7.5**

1. Convenience functions Kotlin has several convenience functions build in:
   1. arrayOf( ) : returns an array containing the specified elements:

arrayOf( a, b, c, … n)

* 1. forEach( ) { }: an enhanced for loop. The loop parameter must be called ***it***.
  2. filter{ condition}. Filter returns an arrayList of elements that satisfy the condition. The test parameter must be called ***it***. An Example:

**v***ar sum = 0*

*arrayOf(-1, 2, -3, 4, -5).filter {* ***it*** *> 0 }.forEach {*

*sum +=* ***it*** *// 🡨this it is for the forEach loop*

*}*

* 1. run{ } . The run function lets you write code that all gets executed with the same calling object. Instead of writing:

*object.name = “Steven”*

*object.age = 30*

you can write:

*object?.run{*

*name = “Steven”*

*age = 30*

*}*

1. Summary Kotlin aims to take some of the tedious boilerplate code out of Java. It is definitely not an introduction language since you need to understand what the boilerplate code is doing first before you can shortcut it. This is a great candidate for upper year students looking to learn another language. There are more examples and tutorials at the official website:

https://kotlinlang.org/docs/reference/