FREE eBook

LEARNING Kotlin

Free unaffiliated eBook created from
Stack Overflow contributors.

#kotlin

Chapter 1: Getting started with Kotlin

Remarks

Kotlin is a statically-typed object-oriented programming language developed by JetBrains primarily targeting the JVM. Kotlin is developed with the goals of being quick to compile, backwards-compatible, very type safe, and 100% interoperable with Java. Kotlin is also developed with the goal of providing many of the features wanted by Java developers. Kotlin's standard compiler allows it to be compiled both into Java bytecode for the JVM and into JavaScript.

Compiling Kotlin

Kotlin has a standard IDE plugin for Eclipse and IntelliJ. Kotlin can also be compiled using Maven, using Ant, and using Gradle, or through the command line.

It is worth noting in `$ kotlinc Main.kt` will output a java class file, in this case `MainKt.class` (Note the Kt appended to the class name). However if one was to run the class file using `$ java MainKt` java will throw the following exception:

```
Exception in thread "main" java.lang.NoClassDefFoundError: kotlin/jvm/internal/Intrinsics
    at MainKt.main(Main.kt)
Caused by: java.lang.ClassNotFoundException: kotlin.jvm.internal.Intrinsics
    at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:335)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
    ... 1 more
```

In order to run the resulting class file using Java, one must include the Kotlin runt-time jar file to the current class path.

```
java -cp .:/path/to/kotlin/runtime/jar/kotlin-runtime.jar MainKt
```

Versions

Version	Release Date
1.0.0	2016-02-15
1.0.1	2016-03-16
1.0.2	2016-05-13
1.0.3	2016-06-30
1.0.4	2016-09-22

Here, Enter two number from the command line to find the maximum number. Output :

```
Enter Two number
71 89 // Enter two number from command line

The value of b is 89
Max number is: 89
```

For !! Operator Please check Null Safety.

Note: Above example compile and run on Intellij.

Read Getting started with Kotlin online: https://riptutorial.com/kotlin/topic/490/getting-started-with-kotlin

Chapter 2: Annotations

Examples

Declaring an annotation

Annotations are means of attaching metadata to code. To declare an annotation, put the annotation modifier in front of a class:

```
annotation class Strippable
```

Annotations can have meta-anotations:

```
    @Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,
AnnotationTarget.VALUE_PARAMETER, AnnotationTarget.EXPRESSION)
    annotation class Strippable
```

Annotations, like other classes, can have constructors:

```
annotation class Strippable(val importanceValue: Int)
```

But unlike other classes, is limited to the following types:

- types that correspond to Java primitive types (Int, Long etc.);
- strings
- classes (Foo:: class)
- enums
- other annotations
- arrays of the types listed above

Meta-annotations

When declaring an annotation, meta-info can be included using the following meta-annotations:

- `@Target`: specifies the possible kinds of elements which can be annotated with the annotation (classes, functions, properties, expressions etc.)
- `@Retention` specifies whether the annotation is stored in the compiled class files and whether it's visible through reflection at runtime (by default, both are true.)
- `@Repeatable` allows using the same annotation on a single element multiple times.
- `@MustBeDocumented` specifies that the annotation is part of the public API and should be included in the class or method signature shown in the generated API documentation.

Example:

```
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,
        AnnotationTarget.VALUE_PARAMETER, AnnotationTarget.EXPRESSION)
@Retention(AnnotationRetention.SOURCE)
@MustBeDocumented
annotation class Fancy
```

Read Annotations online: https://riptutorial.com/kotlin/topic/4074/annotations

Chapter 3: Arrays

Examples

Generic Arrays

Generic arrays in Kotlin are represented by `Array<T>`.

To create an empty array, use `emptyArray<T>()` factory function:

```
val empty = emptyArray<String>()
```

To create an array with given size and initial values, use the constructor:

```
var strings = Array<String>(size = 5, init = { index -> "Item #$index" })
print(Arrays.toString(a)) // prints "[Item #0, Item #1, Item #2, Item #3, Item #4]"
print(a.size) // prints 5
```

Arrays have `get(index: Int): T` and `set(index: Int, value: T)` functions:

```
strings.set(2, "ChangedItem")
print(strings.get(2)) // prints "ChangedItem"

// You can use subscription as well:
strings[2] = "ChangedItem"
print(strings[2]) // prints "ChangedItem"
```

Arrays of Primitives

These types **do not** inherit from `Array<T>` to avoid boxing, however, they have the same attributes and methods.

Kotlin type	Factory function	JVM type
`BooleanArray`	`booleanArrayOf(true, false)`	`boolean[]`
`ByteArray`	`byteArrayOf(1, 2, 3)`	`byte[]`
`CharArray`	`charArrayOf('a', 'b', 'c')`	`char[]`
`DoubleArray`	`doubleArrayOf(1.2, 5.0)`	`double[]`
`FloatArray`	`floatArrayOf(1.2, 5.0)`	`float[]`
`IntArray`	`intArrayOf(1, 2, 3)`	`int[]`
`LongArray`	`longArrayOf(1, 2, 3)`	`long[]`
`ShortArray`	`shortArrayOf(1, 2, 3)`	`short[]`

Extensions

`average()` is defined for `Byte`, `Int`, `Long`, `Short`, `Double`, `Float` and always returns `Double`:

```
val doubles = doubleArrayOf(1.5, 3.0)
print(doubles.average()) // prints 2.25

val ints = intArrayOf(1, 4)
println(ints.average()) // prints 2.5
```

`component1()`, `component2()`, ... `component5()` return an item of the array

`getOrNull(index: Int)` returns null if index is out of bounds, otherwise an item of the array

`first()`, `last()`

`toHashSet()` returns a `HashSet<T>` of all elements

`sortedArray()`, `sortedArrayDescending()` creates and returns a new array with sorted elements of current

`sort()`, `sortDescending` sort the array in-place

`min()`, `max()`

Iterate Array

You can print the array elements using the loop same as the Java enhanced loop, but you need to change keyword from `:` to `in`.

```
val asc = Array(5, { i -> (i * i).toString() })
for(s : String in asc){
    println(s);
}
```

You can also change data type in for loop.

```
val asc = Array(5, { i -> (i * i).toString() })
for(s in asc){
    println(s);
}
```

Create an array

```
val a = arrayOf(1, 2, 3) // creates an Array<Int> of size 3 containing [1, 2, 3].
```

Create an array using a closure

```
val a = Array(3) { i -> i * 2 } // creates an Array<Int> of size 3 containing [0, 2, 4]
```

Create an uninitialized array

```
val a = arrayOfNulls<Int>(3) // creates an Array<Int?> of [null, null, null]
```

The returned array will always have a nullable type. Arrays of non-nullable items can't be created uninitialized.

Read Arrays online: https://riptutorial.com/kotlin/topic/5722/arrays

Chapter 4: Basic Lambdas

Syntax

- Explicit parameters:
- { parameterName: ParameterType, otherParameterName: OtherParameterType -> anExpression() }
- Inferred parameters:
- val addition: (Int, Int) -> Int = { a, b -> a + b }
- Single parameter `it` shorthand
- val square: (Int) -> Int = { it*it }
- Signature:
- () -> ResultType
- (InputType) -> ResultType
- (InputType1, InputType2) -> ResultType

Remarks

Input type parameters can be left out when they can be left out when they can be inferred from the context. For example say you have a function on a class that takes a function:

```
data class User(val fistName: String, val lastName: String) {
    fun username(userNameGenerator: (String, String) -> String) =
        userNameGenerator(firstName, secondName)
}
```

You can use this function by passing a lambda, and since the parameters are already specified in the function signature there's no need to re-declare them in the lambda expression:

```
val user = User("foo", "bar")
println(user.userName { firstName, secondName ->
     "${firstName.toUppercase}"_"${secondName.toUppercase}"
 }) // prints FOO_BAR
```

This also applies when you are assigning a lambda to a variable:

```
//valid:
val addition: (Int, Int) = { a, b -> a + b }
//valid:
val addition = { a: Int, b: Int -> a + b }
```

```
//error (type inference failure):
val addition = { a, b -> a + b }
```

When the lambda takes one parameter, and the type can be inferred from the context, you can refer to the parameter by `it`.

```
listOf(1,2,3).map { it * 2 } // [2,4,6]
```

Examples

Lambda as parameter to filter function

```
val allowedUsers = users.filter { it.age > MINIMUM_AGE }
```

Lambda passed as a variable

```
val isOfAllowedAge = { user: User -> user.age > MINIMUM_AGE }
val allowedUsers = users.filter(isOfAllowedAge)
```

Lambda for benchmarking a function call

General-purpose stopwatch for timing how long a function takes to run:

```
object Benchmark {
    fun realtime(body: () -> Unit): Duration {
        val start = Instant.now()
        try {
            body()
        } finally {
            val end = Instant.now()
            return Duration.between(start, end)
        }
    }
}
```

Usage:

```
val time = Benchmark.realtime({
    // some long-running code goes here ...
})
println("Executed the code in $time")
```

Read Basic Lambdas online: https://riptutorial.com/kotlin/topic/5878/basic-lambdas

Chapter 5: Basics of Kotlin

Introduction

This topic covers the basics of Kotlin for beginners.

Remarks

1. Kotlin file has an extension .kt.
2. All classes in Kotlin have a common superclass Any, that is a default super for a class with no supertypes declared(similar to Object in Java).
3. Variables can be declared as val(immutable- assign once) or var(mutables- value can be changed)
4. Semicolon is not needed at end of statement.
5. If a function does not return any useful value, its return type is Unit.It is also optional.
   6.Referential equality is checked by the === operation. a === b evaluates to true if and only if a and b point to the same object.

Examples

Basic examples

1.The Unit return type declaration is optional for functions. The following codes are equivalent.

```
    fun printHello(name: String?): Unit {
        if (name != null)
            println("Hello ${name}")
    }

    fun printHello(name: String?) {
        ...
    }
```

2.Single-Expression functions:When a function returns a single expression, the curly braces can be omitted and the body is specified after = symbol

```
  fun double(x: Int): Int = x * 2
```

Explicitly declaring the return type is optional when this can be inferred by the compiler

```
  fun double(x: Int) = x * 2
```

3.String interpolation: Using string values is easy.

```
In java:
    int num=10
```

```
    String s  = "i =" + i;

In Kotlin
    val num = 10
    val s = "i = $num"
```

4.In Kotlin, the type system distinguishes between references that can hold null (nullable references) and those that can not (non-null references). For example, a regular variable of type String can not hold null:

```
var a: String = "abc"
a = null // compilation error
```

To allow nulls, we can declare a variable as nullable string, written String?:

```
var b: String? = "abc"
b = null // ok
```

5.In Kotlin,== actually checks for equality of values.By convention, an expression like a == b is translated to

```
   a?.equals(b) ?: (b === null)
```

Read Basics of Kotlin online: https://riptutorial.com/kotlin/topic/10648/basics-of-kotlin

Chapter 6: Class Delegation

Introduction

A Kotlin class may implement an interface by delegating its methods and properties to another object that implements that interface. This provides a way to compose behavior using association rather than inheritance.

Examples

Delegate a method to another class

```
interface Foo {
    fun example()
}

class Bar {
    fun example() {
        println("Hello, world!")
    }
}

class Baz(b : Bar) : Foo by b

Baz(Bar()).example()
```

The example prints `Hello, world!`

Read Class Delegation online: https://riptutorial.com/kotlin/topic/10575/class-delegation

Chapter 7: Class Inheritance

Introduction

Any object-oriented programming language has some form of class inheritance. Let me revise:

Imagine you had to program a bunch of fruit: `Apples`, `Oranges` and `Pears`. They all differ in size, shape and color, that's why we have different classes.

But let's say their differences don't matter for a second and you just want a `Fruit`, no matter which exactly? What return type would `getFruit()` have?

The answer is class `Fruit`. We create a new class and make all fruits inherit from it!

Syntax

- open {Base Class}
- class {Derived Class} : {Base Class}({Init Arguments})
- override {Function Definition}
- {DC-Object} is {Base Class} == true

Parameters

Parameter	Details
Base Class	Class that is inherited from
Derived Class	Class that inherits from Base Class
Init Arguments	Arguments passed to constructor of Base Class
Function Definition	Function in Derived Class that has different code than the same in the Base Class
DC-Object	”Derived Class-Object“ Object that has the type of the Derived Class

Examples

Basics: the 'open' keyword

In Kotlin, classes are **final by default** which means they cannot be inherited from.

To allow inheritance on a class, use the `open` keyword.

```
open class Thing {
    // I can now be extended!
}
```

Note: abstract classes, sealed classes and interfaces will be `open` by default.

Inheriting fields from a class

Defining the base class:

```
open class BaseClass {
    val x = 10
}
```

Defining the derived class:

```
class DerivedClass: BaseClass() {
    fun foo() {
        println("x is equal to " + x)
    }
}
```

Using the subclass:

```
fun main(args: Array<String>) {
    val derivedClass = DerivedClass()
    derivedClass.foo() // prints: 'x is equal to 10'
}
```

Inheriting methods from a class

Defining the base class:

```
open class Person {
    fun jump() {
        println("Jumping...")
    }
}
```

Defining the derived class:

```
class Ninja: Person() {
    fun sneak() {
        println("Sneaking around...")
    }
}
```

The Ninja has access to all of the methods in Person