Kotlin

Ferruccio Damiani

Università di Torino
www.di.unito.it/~damiani

Mobile Device Programming
(Laurea Magistrale in Informatica, a.a. 2018-2019)

Kotlin [https://kotlinlang.org/] [https://developer.android.com/kotlin/]

A modern language (like, e.g., Scala and Swift) with advantages over Java:

- **More concise**: Drastically reduce the amount of boilerplate code.
- **Safer**: Avoid entire classes of errors such as null pointer exceptions.

Moreover:

- Interoperable: Leverage existing libraries for the JVM, Android, and the browser.
- Tool-friendly: Choose any Java IDE or build from the command line.

See also [https://kotlinlang.org/docs/reference/comparison-to-java.html]

Outline

Outline

Coding conventions

A coding style guide about:

- Source code organization
- Naming rules
- Formatting
- Documentation comments
- Avoiding redundant constructs
- Idiomatic use of language features
- Coding conventions for libraries

is available at [https://kotlinlang.org/docs/reference/coding-conventions.html]

Defining packages

A source file may start with a package declaration:

```
package foo.bar

import java.util.*

fun baz() { ... }
class Goo { ... }

// ...
```

- Source files can be placed arbitrarily in the file system.
- All the contents (such as classes and functions) of the source file are contained by the package declared. So, in the example above, the full name of baz() is foo.bar.baz, and the full name of Goo is foo.bar.Goo.
- If the package is not specified, the contents of such a file belong to "default" package that has no name.
- A number of packages are imported into every Kotlin file by default.

Defining functions

Function having two Int parameters with Int return type:

```
fun sum(a: Int, b: Int): Int {
    return a + b
}
```

Function with an expression body and inferred return type:

```
fun sum(a: Int, b: Int) = a + b
```

Function returning no meaningful value:

```
fun printSum(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a + b}")
}
```

Unit return type can be omitted:

```
fun printSum(a: Int, b: Int) {
    println("sum of $a and $b is ${a + b}")
}
```

Default Arguments

```kotlin
fun read(b: Array<Byte>, off: Int = 0, len: Int = b.size) { ... }
```

Named Arguments

```kotlin
fun foo(bar: Int = 0, baz: Int) { ... }

foo(baz = 1) // The default value bar = 0 is used
```

With named arguments we can make the code much more readable:

```kotlin
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    divideByCamelHumps = false,
    wordSeparator = '_'
)
```

and if we do not need all arguments:

```kotlin
reformat(str, wordSeparator = '_')
```

Defining variables

Assign-once (read-only) local variable:

```
val a: Int = 1  // immediate assignment
val b = 2   // ‘Int‘ type is inferred
val c: Int  // Type required when no initializer is provided
c = 3       // deferred assignment
```

Mutable variable:

```
var x = 5 // ‘Int‘ type is inferred
x += 1
```

Top-level variables:

```
val PI = 3.14
var x = 0

fun incrementX() {
    x += 1
}
```

Local Functions (i.e. a function inside another function)

```
fun dfs(graph: Graph) {
    fun dfs(current: Vertex, visited: Set<Vertex>) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v, visited)
    }
    dfs(graph.vertices[0], HashSet())
}
```

Local function can access local variables of outer functions (i.e. the closure), so in the case above, the visited can be a local variable:

```
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()
    fun dfs(current: Vertex) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v)
    }
    dfs(graph.vertices[0])
}
```

Using nullable values and checking for null

A reference must be explicitly marked as nullable when null value is possible.
Return null if str does not hold an integer:

```
fun parseInt(str: String): Int? {
    // ...
}
```

Use a function returning nullable value:

```
fun printProduct(arg1: String, arg2: String) {
    val x = parseInt(arg1)
    val y = parseInt(arg2)

    // Using `x * y` yields error because they may hold nulls.
    if (x != null && y != null) {
        // x and y are automatically cast to non-nullable after null check
        println(x * y)
    }
    else {
        println("either '$arg1' or '$arg2' is not a number")
    }
}
```

or

```
// ...
if (x == null) {
    println("Wrong number format in arg1: '$arg1'")
    return
}
if (y == null) {
    println("Wrong number format in arg2: '$arg2'")
    return
}

// x and y are automatically cast to non-nullable after null check
println(x * y)
```

Generic functions and Variable number of arguments

Generic Functions

```
fun <T> singletonList(item: T): List<T> { ... }
```

Variable number of arguments: at most one parameter of a function (normally the last one) may be marked with vararg modifier:

```
fun <T> asList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts) // ts is an Array
        result.add(t)
    return result
}
```

allowing a variable number of arguments to be passed to the function:

```
val list = asList(1, 2, 3)
```

Inside a function a vararg-parameter of type T is visible as an array of T, i.e. the ts variable in the example above has type Array<out T>.

QUESTION: what is the meaning of the "out" keyword?

Outline

Everything is an object

Everything is an object: we can call member functions and properties on any expression.
QUESTION: what are "member functions" and "properties"?
Some of the types can have a special internal representation (e.g., numbers, characters and booleans can be represented as primitive values at runtime) but to the user they look like ordinary classes.
A presentation about:

- Numbers
- Characters
- Booleans
- Arrays
- Unsigned integers
- Strings

is available at:
`https://kotlinlang.org/docs/reference/basic-types.html`

Representation

On the Java platform, numbers are physically stored as JVM primitive types, unless we need a nullable number reference (e.g. `Int?`) or generics are involved. In the latter cases numbers are boxed.

Note that boxing of numbers does not necessarily preserve identity:

```
val a: Int = 10000
println(a === a) // Prints 'true'
val boxedA: Int? = a
val anotherBoxedA: Int? = a
println(boxedA === anotherBoxedA) // !!!Prints 'false'!!!
```

On the other hand, it preserves equality:

```
val a: Int = 10000
println(a == a) // Prints 'true'
val boxedA: Int? = a
val anotherBoxedA: Int? = a
println(boxedA == anotherBoxedA) // Prints 'true'
```

Explicit Conversions

Due to different representations, smaller types are not subtypes of bigger ones. If they were, we would have troubles of the following sort:

```
// Hypothetical code, does not actually compile:
val a: Int? = 1 // A boxed Int (java.lang.Integer)
val b: Long? = a // implicit conversion yields a boxed Long (java.lang.Long)
print(b == a) // Surprise! This prints "false" as Long's equals() checks whether the
      other is Long
```

Smaller types are NOT implicitly converted to bigger types. This means that we cannot assign a value of typeByte to an `Int` variable without an explicit conversion

```
val b: Byte = 1 // OK, literals are checked statically
val i: Int = b // ERROR
```

We can use explicit conversions to widen numbers

```
val i: Int = b.toInt() // OK: explicitly widened
print(i)
```

Arrays

Arrays are represented by the Array class, that has get and set functions (that turn into [] by operator overloading conventions), and size property, along with a few other useful member functions:

```
class Array<T> private constructor() {
    val size: Int
    operator fun get(index: Int): T
    operator fun set(index: Int, value: T): Unit

    operator fun iterator(): Iterator<T>
    // ...
}
```

To create an array, we can use a library function arrayOf() and pass the item values to it, so that arrayOf(1, 2, 3) creates an array [1, 2, 3]. Alternatively, the arrayOfNulls() library function can be used to create an array of a given size filled with null elements.

Another option is to use the `Array` constructor that takes the array size and the function that can return the initial value of each array element given its index:

```kotlin
// Creates an Array<String> with values ["0", "1", "4", "9", "16"]
val asc = Array(5, { i -> (i * i).toString() })
asc.forEach { println(it) }
```

Unlike Java, **arrays in Kotlin are invariant.** This means that Kotlin does not let us assign an `Array<String>` to an `Array<Any>`, which prevents a possible runtime failure (but you can use `Array<out Any>`.

QUESTION: what is `Any`?

Kotlin also has specialized classes to represent arrays of primitive types without boxing overhead: `ByteArray`, `ShortArray`, `IntArray` and so on. These classes have no inheritance relation to the Array class, but they have the same set of methods and properties. Each of them also has a corresponding factory function:

```
val x: IntArray = intArrayOf(1, 2, 3)
x[0] = x[1] + x[2]
```

Strings

Strings are represented by the type String. Strings are immutable. Elements of a string are characters that can be accessed by the indexing operation: `s[i]`. A string can be iterated over with a for-loop:

```
for (c in str) {
    println(c)
}
```

Escaping is done in the conventional way, with a backslash.
A raw string is delimited by a triple quote (`"""`), contains no escaping and can contain newlines and any other characters:

```
val text = """
    for (c in "foo")
        print(c)
"""
```

String Templates

Strings may contain template expressions, i.e. pieces of code that are evaluated and whose results are concatenated into the string. A template expression starts with a dollar sign ($) and consists of either a simple name:

```
val i = 10
println("i = $i") // prints "i = 10"
```

or an arbitrary expression in curly braces:

```
val s = "abc"
println("$s.length is ${s.length}") // prints "abc.length is 3"
```

Templates are supported both inside raw strings and inside escaped strings. If you need to represent a literal $ character in a raw string (which doesn't support backslash escaping), you can use the following syntax:

```
val price = """
${'$'}9.99
"""
```

Outline

Classes and Primary constructors

Classes in Kotlin are declared using the keyword `class`:

```
class Invoice { ... }
class Empty
```

A class can have a **primary constructor** and one or more **secondary constructors**. The primary constructor is part of the class header: it goes after the class name (and optional type parameters).

```
class Person constructor(firstName: String) { ... }
```

If the primary constructor does not have any annotations or visibility modifiers, the `constructor` keyword can be omitted:

```
class Person(firstName: String) { ... }
```

QUESTION: what are the available “annotations” and “visibility modifiers”?

Primary constructors and Initializers

The primary constructor cannot contain any code. Initialization code can be placed in **initializer blocks**, which are prefixed with the `init` keyword.
During an instance initialization, the initializer blocks are executed in the same order as they appear in the class body, interleaved with the property initializers:

```
class InitOrderDemo(name: String) {
 val firstProperty = "First property: $name".also(::println)
 init {
  println("First initializer block that prints ${name}")
 }
 val secondProperty = "Second property: ${name.length}".also(::println)
 init {
  println("Second initializer block that prints ${name.length}")
 }
}
```

QUESTION: what is the “also” method?

During the instance initialization the following text is printed:

```
First property: hello
First initializer block that prints hello
Second property: 5
Second initializer block that prints 5
```

Note that parameters of the primary constructor can be used in the initializer blocks. They can also be used in property initializers declared in the class body:

```
class Customer(name: String) {
    val customerKey = name.toUpperCase()
}
```

In fact, for declaring properties and initializing them from the primary constructor, there is a concise syntax:

```
cllass Person(val firstName: String, val lastName: String, var age: Int) { ... }
```

Much the same way as regular properties, the properties declared in the primary constructor can be mutable (`var`) or read-only (`val`).

Secondary constructors

The class can also declare secondary constructors, which are prefixed with `constructor`:

```
class Person {
    constructor(parent: Person) {
        parent.children.add(this)
    }
}
```

If the class has a primary constructor, each secondary constructor needs to delegate to the primary constructor, either directly or indirectly through another secondary constructor(s). Delegation to another constructor of the same class is done using the `this` keyword:

```
class Person(val name: String) {
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

Note that code in initializer blocks effectively becomes part of the primary constructor. Delegation to the primary constructor happens as the first statement of a secondary constructor, so the code in all initializer blocks is executed before the secondary constructor body. Even if the class has no primary constructor, the delegation still happens implicitly, and the initializer blocks are still executed:

```
class Constructors {
    init {
        println("Init block")
    }

    constructor(i: Int) {
        println("Constructor")
    }
}
```

If a non-abstract class does not declare any constructors (primary or secondary), it will have a generated primary constructor with no arguments. The visibility of the constructor will be public. If you do not want your class to have a public constructor, you need to declare an empty primary constructor with non-default visibility:

```
class DontCreateMe private constructor () { ... }
```

NOTE: On the JVM, if all of the parameters of the primary constructor have default values, the compiler will generate an additional parameterless constructor which will use the default values.

```
class Customer(val customerName: String = "")
```