Fundamentals

From a simple “Hello world” program towards more complex examples, we will be discovering Kotlin features and functionalities. We will try to touch not just the fundamentals of Kotlin but the fundamentals (and history) of JVM (and Java) in general.

In this section, we will introduce you to the very basics of JVM programming, Java’s short history, and move you in direction of Kotlin basic syntax and data type fundamentals. This time we will write some Kotlin code for real. Prepare yourself to dive deep into the programming world of Kotlin!

A short history of Java

As you probably already know Kotlin is a JVM language and it has strong ties with the Java programming language because it is based on Java (JVM) technology. In this section, we will spend some time on a brief history of Java with a focus on features that Java brought through its evolution.

Java is an old programming language. Java appears the first time in the year 1995. Since then up to this day it has grown and achieved great success in the software engineers community. Java is still the most popular language for software development!

For the last 25 years, Java has had several major versions. Each version brought some critical major improvements and a set of features to the language. At the time of writing this book, Java 13 has been released. However, as you will see soon for some significant periods Java wasn't bringing new and significant features to the language. There were years without any new significant releases and no new features to bring to the table.

Let's take a look at the list of significant Java releases and improvements that have been brought to the language:

Java version 5: released in 2004, this is the version of Java where modern Java started to emerge. In Java 5 version the following features have been introduced:

- For-each loop
- Varargs
- Static Import
- Autoboxing and Unboxing
- Support for Enums
- Covariant Return Type
- Annotations
- Generics

Java version 6: released in 2006, this version of Java extends what we consider today the modern Java by introducing:

- Collections framework
- I/O support
- JAR files

- Reflection
- Serialization of Objects

and many more features.

Java version 7: released in 2011, **after 5 whole years of waiting** brings the following set of features:

- Caching multiple Exceptions by single catch
- Support for Strings in switch statements
- Binary Literals
- The try-with-resources
- Underscores in Numeric Literals

Java version 8: released in 2014 (**3 years later!**) brings a set of features that are still being used by most IT companies who use Java:

- Lambda Expressions
- Function references
- Functional Interfaces
- forEach function
- Type annotations and repeating annotations
- Optional class
- Base64 Encode / Decode

and many other features.

Meanwhile, Kotlin reached maturity status as a programming language and many developers switched to Kotlin as its development option no. 1. Kotlin brought cutting-edge features, the lifecycle of a more frequent release, and great flexibility. Since then Java has been released in a couple more versions supporting some of the features that Kotlin already had:

Java version 9: released in 2017 brings a couple of important things:

- Module system
- JShell
- Private Interface functions
- HTTP/2 support

Java versions 10 to 15: released between 2018 and 2020: brings new modern features to the table and more frequent versions release cycle. We will highlight some of the new features:

- New APIs
- Local-Variable Type Inference
- Removed the Java EE and CORBA modules
- Switch expression
- Smart cast
- Multi-line texts
- Records
- Hidden classes
- The Z Garbage Collector (ZGC)

By bringing new features in versions after Java 10, Java started to run side by side with JVM languages such as Kotlin, Groovy, and Scala. It will be interesting how Java will compete with its main rivals in upcoming years. Most likely, we will cover that in the 4th edition of the Fundamental Kotlin book.

What is Java?

What is Java? Let's be more precise in defining Java as a programming language and concept of JVM.

Java is a powerful, general-purpose programming language. Java as we mentioned in the previous section exists since 1995. It comprises the Java programming language, and the Java Virtual Machine (JVM). So, we distinguish two separate parts: language itself and virtual environment (JVM). In the case of Kotlin, it is Kotlin as a programming language and JVM as a running environment for Kotlin programs. A similar philosophy has been adopted by Microsoft for its .NET technology.

Java's ecosystem is a standardized environment controlled and maintained by Oracle Corporation (https://www.oracle.com/). Thanks to these standards developers and consumers are confident that the technology will be compatible with other components, even if they come from different technology vendors.

It is important to note that since Java version 11 Java (Java Development Kit) is commercially licensed! For free use of Java OpenJDK (https://openjdk.java.net/) port is available. OpenJDK (Open Java Development Kit) is open-source implementation of the Java platform. The implementation is licensed under the GNU General Public License (GNU GPL) version 2 with a linking exception.

Java is considered to be easy to read and write. Java has solid grammar and a simple program structure. Java is based on experience with languages like C and C++. Compared to C and C++ Java simplifies its features and makes it easier for every-day use for the developers. The syntax of Java is very similar to C and C++, but it does not have focus on a low-level programming.

Java programming language is object-oriented, class-based, and designed to have as few dependencies as possible. The main philosophy behind Java is that developers write code once and run that code everywhere (thanks to JVM). Java programs are compiled to bytecode that can run on any JVM regardless of the underlying computer architecture.

As we mentioned in the previous section, Java has been released in 14 major releases for the last 25 years bringing the joy of everyday development to more than **9 million** developers who use it every day!

Java Runtime Environment

Java Runtime Environment (JRE), is a set of the minimum components necessary to create and run Java programs and it is a part of a Java Development Kit (JDK).

JRE is made up of the Java Virtual Machine (JVM), Java class libraries, and the Java class loader. The purpose of JDK is to help developers to write Java programs, while JRE has the purpose of only running it.

What is JVM?

We can think of JVM as some kind of computer within a computer. The purpose of JVM is to execute JVM programs. Java, Groovy, Kotlin, and other JVM languages (its programs) are all executed on JVM.

The main characteristic of all JVM languages is that its source code gets compiled to Java bytecode, and then that bytecode is loaded and executed by JVM. Thanks to this, developers do not need to write different codes for different platforms. All platform-specific things are handled by JVM itself. Each particular host operating system needs its implementation of the JVM and Java runtime environment. Each of them interprets bytecode the same way even though its implementations may be different due to platform specificities.

During program execution, JVM performs garbage collection. Garbage collection is the process by which JVM performs automatic memory management for running programs. All unused objects in memory of JVM used by these programs are released (cleaned up) at a specific point in time.

JVM is precisely specified by the specification that ensures interoperability of JVM programs. The garbage collection algorithm used by JVM and any internal optimization of JVM instructions is not specified.

It is interesting to note that for Android compiler converts source code written in Java or Kotlin into bytecode for the Android Runtime (ART). Android does not have classic JVM. JVM and ART work in entirely different ways.

What is the Java ecosystem?

Three basic components that we mentioned until this moment make up the Java ecosystem and they are:

- Java Virtual Machine (JVM)
- Java Runtime Environment (JRE)
- Java Development Kit (JDK)

These components are core parts that are shipped by Java implementations. In the next section, we will check out how Kotlin fits into this.

How Kotlin relates to Java?

Kotlin is a JVM language. Same way as with Java, we will produce bytecode from our source code. As we already mentioned Java bytecode is executed equally on all versions of the JVM platform independently from the platform itself.

Kotlin is completely interoperable with Java. You can easily call Kotlin code from Java and Java code from Kotlin. Thanks to this adopting Kotlin in the existing ecosystem is much easier. As we demonstrated in the previous section of the book, the creators of Kotlin provided us with a tool for direct code conversion. It is really easy to convert existing java source code into Kotlin.

Lifecycle of the program

Each program that we run has its entry point. The entry point for every JVM program is the “main” function. Kotlin typical “main” function looks like this:

```
package net.milosvasic.fundamental.kotlin

fun main(args: Array<String>) {

    // Your program starts from here
}
```

Arguments can be left out:

```
package net.milosvasic.fundamental.kotlin

fun main() {

    // Your program starts from here
}
```

The main function in Kotlin must be placed in ordinary .kt file under the package. Arguments (“args: Array<String>”) contain everything that we pass to our program from the command line (terminal). This data can be used later as parameters in our programs. You will learn more about function arguments in upcoming sections of this book. When the main function is entered execution of our program begins.

After the program finishes with the execution proper exit code is returned as the result. The default return value of every JVM program is zero. Zero means that the program has been executed with success. Non-zero codes mean that our program had abnormal termination. Non-zero values can be positive and negative. Positive values are usually returned from our code (defined by the user) to indicate a particular exception. Negative status codes are system-generated error codes. Such error codes are generated as a result of unanticipated exceptions, system errors, or forced termination of our program.

Take a look at file “SystemExitSuccess.kt” form book’s code examples:

```
package net.milosvasic.fundamental.kotlin.lifecycle

import kotlin.system.exitProcess

fun main() {
    println("Exiting with success")
    exitProcess(0)
}
```

This program prints a message and exits with zero, meaning that it is completed with success. “exitProcess” function terminates the program and returns 0 as a result of its execution. Now open “SystemExitFailure.kt” example file:

```
package net.milosvasic.fundamental.kotlin.lifecycle

import kotlin.system.exitProcess

fun main() {

    println("Something went wrong!")
    exitProcess(1)
}
```

Besides different message that is printed out, this programs returns “1” as execution result. This means that the program has finished with failure.

Basic syntax

Since we have introduced you to basic Java legacy and explained to you some basics of Kotlin (JVM) programs it is time to go further with Kotlin basics. It is time for us to start with Kotlin programming language syntax.

Programming language syntax refers to the spelling and grammar of a language. Computers are different than people. They will understand what you type only if you type it in the exact form that the computer expects. This form is called the programming language syntax. Every programming language has its syntax. Some of them are similar, but some of them are quite different. Kotlin has similarities with multiple programming languages by which designers of the Kotlin programming language have been inspired. The most obvious syntax similarity comes from Java. However, some other languages influenced the design and syntax of Kotlin:

- Scala: https://www.scala-lang.org/
- Groovy: https://groovy-lang.org/