Fundamental Kotlin revised

Милош Васиђ
(latin: Miloš Vasić)

Third Edition
www.fundamental-kotlin.com

Fundamental Kotlin revised by Miloš Vasić
Third Edition

Published by Miloš Vasić, i@mvasic.ru
ISBN: 978-86-920307-2-7
Website: www.fundamental-kotlin.com

Table of Contents

About “Fundamental” book serial

A Fundamental book serial is a book serial to bring readers quickly into the book's subject and make it possible to relatively easy start working with the matter. Every book from serial assumes that the reader has at least some basic knowledge of computer programming and computer technology in general. Fundamental Kotlin revised is the first book in Fundamental serial released in three editions.

What is different compared to the Second Edition?

In this edition of the book, a lot of it has been added. Book has a new structure and new sections are introduced. In this edition of the book, the author tried to bring more material to better illustrate Kotlin's everyday use.

The book covers Kotlin programming language version **1.7.10**. With that in mind author focused on bringing the latest and the greatest features of the Kotlin programming language to the readers.

What is this book about?

Fundamental Kotlin revised is a book focused on Kotlin programming language, on language's most important features and aspects. All examples and the code for this book are located on the GitHub repository:

https://github.com/milos85vasic/Fundamental-Kotlin/releases

For all ideas and questions please contact the author by one of the contact options listed in the "About the author" section of this book.

Who is this book for?

This book is for people who are willing to try something new. Fundamental Kotlin revised (third edition of Fundamental Kotlin book serial) is imagined not just as a guide to Kotlin for experienced developers, but also for students or technology enthusiasts. So, if you are not a senior developer it should not be a problem. However, some fundamentals of computing are assumed. The reader should be familiar with some programming fundamentals, with git basics and bash shell basic commands.

Fonts used in this book

This book uses two typefaces: Ubuntu and JetBrains Mono. Both fonts have been used for all publication versions of the book.

Ubuntu

JetBrains Mono

About the author

Miloš Vasić is a software engineer from Belgrade, Serbia. For most of his career, he was working as an Android software engineer. Miloš is also well-skilled in other technologies and other programming languages besides Kotlin and Android.

Miloš is passionate about software development and ways to make software better. He prefers to use his software if it is possible and to avoid introducing unknown code of third parties into projects. Miloš is working constantly on his code, various projects like software components or small programs.

When he is not developing software, he is spending time learning and investigating new technologies. One such technology was Kotlin. Miloš chose Kotlin as his primary development language because it gave him everything that he needs to achieve great flexibility and product quality.

You can reach the author from one of the following items:

Email: milos85vasic@gmail.com
Linkedin: miloš-vasić-53778682
Xing: Milos_Vasic4
Github: milos85vasic
Website: www.milosvasic.net

Preface

Today Kotlin is the main programming language for Android development. Thanks to its popularity Kotlin became the language for any other JVM development. For example, many backend developers are using Kotlin for Spring application programming. We are all witnesses of the increasing number of Kotlin communities, conferences and workshops that have been held, many books that have been written, and demand for various software components. Many of these components have been ported from Java and some of them continued their development only in pure Kotlin.

Kotlin is powerful. Everything that we did in Java can be raised to a completely new level. Kotlin is a tool for everything, a general-purpose first-class programming language. What you will get from Kotlin is less stress, fever bugs, and more joy while working the best job in the world, begin software engineer!

Where Kotlin can be used? You can use Kotlin to develop your web server, to write various programming libraries, or for example to write plugins. To get into Kotlin read this book carefully, use internet resources, and most importantly, write as much code as you can!

What is Kotlin?

Kotlin is a statically typed programming language running on Java Virtual Machine, Android, web browser, or even native as a binary.

Kotlin is **concise:** drastically reduces source code boilerplate. It is **safe:** (almost) impossible to get NullPointerException (NPE). We will learn about NPEs in upcoming sections of this book. Kotlin is **versatile:** it is a general-purpose programming language. Finally, it is **interoperable**. That means that we can use existing JVM libs and frameworks in Kotlin, or use Kotlin developed libraries in other JVM languages.

A team at JetBrains (creators of IntelliJ Idea IDE) developed Kotlin, an open-source language with an army of external contributors.

Kotlin is licensed under Apache 2 Open-Source License. The current version of Kotlin is **1.7.10**.

Some basic concepts explained

Before we start with Kotlin let's explain a couple of terms that we have mentioned in the previous section just in case that you are not yet familiar with them.

What is a statically typed programming language?

We said that Kotlin is a statically typed language. We will explain the meaning of statically typed programming languages in comparison between statically typed and dynamically typed programming languages. Dynamically typed programming languages perform type checking on runtime. On the other hand, statically typed languages such as Java and Kotlin perform type checking at compile time. In Kotlin as an example, variables must be declared before values are assigned to them.

What is Java Virtual Machine or JVM?

If you are not experienced with Java, let's take a moment to explain what JVM (Java Virtual Machine) is. JVM is software running on your system (engine) that is responsible for running all Java software. JVM converts Java bytecode (we will talk more about it soon) into the machine language. Thanks to this JVM applications are the same on all platforms.

Software developers write JVM applications once for all platforms. That code is executable on every operating system. All specificity for the particular operating system is handled by the version of JVM installed for that system. So, we have installations of Java for Linux, macOS, Microsoft Windows, and so on. It is also worth mentioning that JVM is part of JRE (Java Runtime Environment).

Kotlin fits into this as JVM language, which means, all code that we write in Kotlin is compiled into Java bytecode and executed on JVM (Java Virtual Machine).

Java bytecode

We will spend one brief moment explaining the meaning of Java bytecode. Java bytecode represents the instruction set for JVM. Our source code of the program is translated into Java bytecode during compilation by the Java compiler (Kotlin has its compiler for this purpose). Bytecode is then loaded into JVM and its instructions are executed.

The lifecycle of a typical JVM program looks like this:

- source code for the program
- compiler creates Java bytecode from program source code
- JVM loads bytecode that has been created by the compiler
- JVM converts bytecode into machine code (language)
- instructions are executed by computer hardware.

Note:

If you are not familiar with Java at all we will explain some of its concepts in the "Legacy of Java" section of this book.

Basic characteristics of Kotlin

Kotlin has many powerful characteristics. We will highlight some of the most important ones.

As we previously mentioned **Kotlin runs on JVM** (Java Virtual Machine) meaning that it is cross-platform compatible. For you who don't know, Kotlin is also available in its **Kotlin native** and **JavaScript** flavors. That practically means that we can write the code that will be compiled directly for the execution on computer hardware or we can write Kotlin programs for the web.

Kotlin is **statically typed**, the type of a variable is known at compile time.

Kotlin is **functional** too. With first-class functions, you can store functions as variables or pass as parameters to other functions as parameters.

Immutability: It is guaranteed that the state of an immutable object can't change over time.

Kotlin has **coroutines**. Thanks to coroutines support Kotlin makes your life easier when comes to asynchronous or non-blocking programming. We will cover coroutines in a separate section of this book. Other previously mentioned features will be covered in separate sections as well.

And last but not least, Kotlin is an **object-oriented programming language** that is free and open source. If you are interested to contribute to Kotlin development you can check out (or fork) GitHub repository:

https://github.com/JetBrains/kotlin.

Where is it used?

As a general-purpose programming language Kotlin can satisfy the needs of every professional software developer. We will cover some use-cases of Kotlin.

Server-side development

Kotlin is great for developing server-side applications. Kotlin allows you to write concise and expressive code while maintaining full compatibility with existing Java-based technology stacks and a smooth learning curve.

So, what are the benefits?

- Expressiveness
- Scalability
- Interoperability
- Easy migration
- Great tooling
- Easy learning Curve.

This is the list of some web framework that you can try for web development:

Ktor, https://ktor.io
Spring, https://spring.io
Vert.x, http://vertx.io
Kotlin DSL for HTML, https://github.com/kotlin/kotlinx.html
Wasabi, https://github.com/hhariri/wasabi
Hexagon, https://github.com/jaguililla/hexagon

and many others.

Android mobile development

Kotlin is the main programming language for writing Android applications. It is unimaginable by many Android developers to do their regular programming in anything different than Kotlin. To find out more about Android you can visit the official Android developer's page:

https://developer.android.com.

On market, some tools go beyond the standard language features. One of them is Android KTX, the set of Kotlin extensions for Android development:

https://developer.android.com/kotlin/ktx.

JavaScript development

Kotlin provides us with the ability to produce JavaScript builds. It does so by translating Kotlin source code to JavaScript. The current implementation targets ECMAScript 5.1 but there are plans to eventually target ECMAScript 2015 too.

When you choose the JavaScript target, any Kotlin code that is part of the project as well as the standard library that ships with Kotlin are translated to JavaScript. This excludes the JDK and any JVM or Java framework or library used. Any file that is not Kotlin will be ignored during compilation.

Native development

Kotlin can be compiled to run directly on computer hardware. For this purpose team at JetBrains developed Kotlin/Native technology that is developed side by side with the main Kotlin language. It is really easy to write Kotlin applications that will be delivered as native platform binaries. We will talk more about Kotlin/Native in upcoming sections.

Kotlin for data science

Kotlin found its place even in data science. Kotlin supports integration with some very popular platforms used by data scientists. We will mention some of the most popular.

Apache Zeppelin (comes shipped with Kotlin interpreter):

https://zeppelin.apache.org/

Project Jupyter:

https://jupyter.org/
https://github.com/Kotlin/kotlin-jupyter.

It is also worth mentioning that because of Kotlin’s huge popularity developers community created a significant number of great libraries for data-related tasks. We will list some of them.

JetBrains’s Lets Plot:

https://github.com/JetBrains/lets-plot, an open-source plotting library for statistical data.

Kmath:

https://github.com/mipt-npm/kmath, a Kotlin-based analog to Python's “numpy” library. In contrast to ”numpy” and ”scipy” it is modular and has a lightweight core.

Kotlin Statistics:

https://github.com/thomasnield/kotlin-statistics, collection of helpful extension functions to perform exploratory and production statistics in a Kotlin-idiomatic way.

Krangl:

https://github.com/holgerbrandl/krangl, Kotlin library for data wrangling.

Kravis:

https://github.com/holgerbrandl/kravis, a Kotlin grammar for data visualization.

okAlgo:

https://github.com/optimatika/okAlgo, Idiomatic Kotlin extensions for ojAlgo (https://www.ojalgo.org/), with some inspirations from PuLP (https://github.com/coin-or/pulp).

Data2viz:

https://data2viz.io/, data visualization.

Sparklin:

https://github.com/khud/sparklin, Kotlin language support for Apache Spark.

Koma:

https://github.com/kyonifer/koma, a scientific computing environment for Kotlin.

Komputation:

https://github.com/sekwiatkowski/komputation, a neural network framework for the JVM written in Kotlin and CUDA C.

KotlinNLP:

https://github.com/KotlinNLP, natural language processing in Kotlin.

Jinx:

https://exceljava.com/docs/tutorials/kotlin.html, an Excel Add-In that enables developers to extend Excel’s capabilities.

Kotlin Algorithm:

https://github.com/gazolla/Kotlin-Algorithm, implementations of popular algorithms and data structures including machine learning.

Building programs

Kotlin supports several major technologies: Kotlin running on JVM, Android, JavaScript, or native. To generate the final build from your source code it is required to have the Kotlin compiler installed. With each Kotlin release, the standalone version of the compiler is shipped. We can download it from GitHub:

https://github.com/JetBrains/kotlin/releases

Unzip the standalone compiler into a directory and optionally add the bin directory to your system path. The bin directory contains the scripts needed to compile and run Kotlin on Windows, macOS, and Linux.

Kotlin compiler can be also installed in several other ways.

Installing Kotlin compiler

Below is a couple of nice little ways to install the Kotlin compiler on your system. Chose the way that most suits you.

SDKMAN

An easier way to install Kotlin on UNIX-based systems such as macOS, Linux, Cygwin, FreeBSD, and Solaris is by using SDKMAN. Simply run the following in a terminal and follow instructions:

```
$ curl -s https://get.sdkman.io | bash
```

Next, open a new terminal and install Kotlin by executing the following command:

```
$ sdk install kotlin
```

Homebrew

Alternatively, on macOS, you can install the compiler via Homebrew.

```
$ brew update
$ brew install kotlin
```

MacPorts

If you're a MacPorts user, you can install the compiler with:

```
$ sudo port install kotlin
```

Compiling Kotlin source code

Let’s see now how the build process works in Kotlin.

All source code is stored in .kt files that are organized in packages (filesystem directories organized hierarchically). Kotlin compiler analyzes source code and generates .class files. Then, .class files are packaged. How they are packaged depends on the kind of project you are working on.

The following example will demonstrate how we can compile Kotlin source code and execute compiled programs.

Create simple file named “First.kt” with the following content:

```
package net.milosvasic.fundamental.kotlin

fun main(args: Array<String>) {

    println("My first Kotlin application.")
}
```

The first line of the code tells us under which package resides our program. More about packages will be discussed in upcoming sections of the book. Then, we defined the main program function. The main function and functions, in general, will be discussed soon as well. For now, it is important to know that main function is the entry point in the execution of our program. Inside the body of our main function (body of the program), we are executing “println” function that will print a simple message on our screen.

In two consecutive commands we will compile and run our code from the source code .kt file:

```
$ kotlinc First.kt -include-runtime -d First.jar
$ java -jar First.jar
```

The output of the execution is:

```
$ My first Kotlin application.
```

Kotlin to JavaScript

The following example illustrates how Kotlin compiles for JavaScript.

Clone Fundamental Kotlin examples from GitHub repository. Open your terminal and cd to the JavaScript directory. Inside the directory there is file JsExampleLibrary.kt located with the following content:

```
package net.milosvasic.fundamental.kotlin.javascript

fun helloJS() {

    println("Hello from JavaScript!")
}
```

Compile the library using the JavaScript compiler:

```
$ kotlinc-js -output library.js -meta-info JsExampleLibrary.kt
```

After compilation we will have two new files:

```
library.js
library.meta.js
```

You can simply distribute two .js files: library.js and library.meta.js. The former file contains translated JavaScript code, the latter file contains some meta-information about Kotlin code, which is needed by the compiler.

As a possible option, you can append the content of library.meta.js to the end of library.js. The resulting file can then be distributed alone.

Also, you can create an archive, which can be distributed as a library:

```
$ jar cf library.jar *.js
```

Using library

Let's see how to use this library. In same directory locate file JsExampleLibraryUse.kt:

```
package net.milosvasic.fundamental.kotlin.javascript
```

```
fun helloJS(count: Int) {
    for (x in 0..count) {
        helloJS()
    }
}
```

Then, compile it with the library that we just created:

```
$ kotlinc-js -output use.js -libraries library.meta.js \ JsExampleLibraryUse.kt
```

“use.js” file is available after compiling.

Both files”library.js” and “library.meta.js” should be present, because translated JavaScript file contains meta-information about inlining, which is needed by the compiler.

If you have an archive “library.jar”, which contains “library.js” and ”library.meta.js”, you can use the following command:

```
$ kotlinc-js -output use.js -libraries library.jar \ JsExampleLibraryUse.kt
```

Kotlin/Native

For cutting-edge performance, you may want to consider the use of Kotlin/Native. Kotlin/Native is a technology developed by JetBrains that makes it possible to compilie Kotlin source code into native binaries.

Installing Kotlin/Native

The latest version of Kotlin/Native at the time of writing this book is **1.7.10**. To install Kotlin/Native download the proper version from the releases page:

https://github.com/JetBrains/kotlin-native/releases

Once the proper archive is downloaded, extract it to the desired directory. To be able to use Kotlin/Native it is required to build it. To do so you must satisfy certain requirements. You must install Java JDK (Java Development Environment) instead of Java JRE (Java Runtime Environment). If you are using macOS you must have Xcode version **11.5** installed on your system. For Linux users, it is required to have “ncurses-compat-libs” installed. On RedHat based systems such as Fedora or CentOS installation is performed by executing the following command:

```
$ yum install ncurses-compat-libs
```

Debian based distributions can install “ncurses-compat-libs” by executing the following command:

```
$ apt install libncurses5
```

Once requirements are met we are ready to build Kotlin/Native. Cd into the directory where you have extracted Kotlin/Native and update dependencies:

```
$ ./gradlew dependencies:update
```

It will take some time for dependencies to be updated. Please wait until the process is complete. It may take a couple of minutes for this step. Then, start the build:

```
$ ./gradlew bundle
```

The time needed for the build to complete may take more than one hour! After the build is done add your Kotlin/Native “bin” directory to the system path. After that, you will be ready to compile your first Kotlin/Native program.

Compiling to native

In this section, we will demonstrate Kotlin/Native compiling. If you didn’t clone book examples, please do so.

Cd into ”Native” directory. Inside you will see the file called ”HelloNative.kt”. The content of the file is almost the same as “First.kt” from our first compiling example:

```
fun main(args: Array<String>) {

    println("My first Kotlin/Native application.")
}
```

You may notice that **we did not define the package** for this example. Let’s compile it. Open a terminal and execute the following command to compile .kt file:

```
$ kotlinc HelloNative.kt -o hello -opt
```

Flag at the end “-opt” stands for optimized compilation. Now run “hello” program:

```
$ ./hello
```

The output of execution will be:

```
My first Kotlin/Native application.
```

Build automation tools

Compiling Kotlin programs by directly executing compiler in practical everyday development is not something that software developers are doing really. For purpose of building and packaging our programs, various build automation tools are available. To make our lives easier build automation tools are automating the creation of executable applications from the source code.

What is automated?

Build automation tools prepare and download all dependencies used by our projects. For example, all libraries are downloaded and stored for later use. The user does not have to worry about providing dependency files. Build automation tools perform all custom scripted tasks that users defined such as copying files or any project-specific activities. Tasks can be executed before or after compiling the project.

Build automation tools perform compiling and proper packaging of our projects, executing tests, and finally deploying projects to production systems.

As we already mentioned you can build your Kotlin code using the Kotlin compiler. However, it is more convenient to use one of the most popular build automation tools.

To build Kotlin projects you can use various build automation tools that support Kotlin out of the box. Let’s mention some of them.

Gradle

https://gradle.org/

Gradle is a build automation tool for multi-language software development. Kotlin is one of the languages that Gradle supports. Gradle controls the development process in the tasks of compiling, packaging, testing, deployment, and publishing.

Gradle is licensed under Apache License 2.0 and it is written pure Java, Groovy, and Kotlin.

Maven

https://maven.apache.org/

Maven is a build automation tool. It is used primarily for Java development. However, Maven supports projects written in C#, Ruby, Scala, and others. The Apache Software Foundation hosts the Maven project:

https://www.apache.org/

Maven was there formerly part of the Jakarta Project. Maven is licensed under Apache License 2.0 and it is written in Java.

Apache Ant

http://ant.apache.org/

Apache Ant is another Apache Foundation project licensed under Apache License 2.0. Apache Ant is a tool for automating software build processes. It originates from the Apache Tomcat project. Apache Ant is developed as a replacement for the Make build tool. It has similarities with Make, however, it is implemented using pure Java language and requires the Java platform to run.

Griffon

http://griffon-framework.org/

Griffon is a desktop application development platform for the JVM inspired by the Grails framework. Griffon is written in Java and licensed under Apache

License 2.0 license. Griffon supports other programming languages such as Groovy and Kotlin:

https://github.com/griffon/griffon-kotlin-plugin

Griffon Kotlin plugin enables compiling and running Kotlin code on Griffon applications. The plugin is written in Groovy and Java and it is licensed under Apache License 2.0 as Griffon framework itself.

Kobalt

http://beust.com/kobalt/home/index.html

Kobalt is a build automation tool written in pure Kotlin. Kobalt is inspired by Gradle and Maven's build automation tools. While takes some good ideas of them Kobalt also provides some new powerful features on its own. As all previously mentioned build automation tools it is licensed under Apache License 2.0.

Creating Kotlin project

In this section, we will cover examples of how to create your Kotlin project. The first approach that we will present to you is quick and easy by using the IntelliJ Idea IDE project creation wizard. The second approach will be more complex, it will present to you how to create a Kotlin project from scratch using Gradle build automation tool. With the latter option, you will have more work to do, but also you will have more control over your project configuration, having more knowledge about files versioned in your code repository and what they are doing exactly.

Creating IDE project

Previous examples were built by using only the Kotlin compiler from the terminal. In the rest of the book, we will be using IntelliJ Idea IDE (an integrated development environment), developed by creators of the Kotlin, JetBrains company.

You can download and install IntelliJ Idea Community Edition. Community Edition of IntelliJ Idea is the free version of the IDE. When it is installed, open preferences and install support for Kotlin programming language if it is not installed already. After this is done, you are ready to write and run some Kotlin code.

To create a new Kotlin project in IntelliJ IDEA do the following:

File → New → Project

Then choose:

Kotlin → JVM | IDEA → Next

Name the project, set the file system location, and choose the Java version from a dropdown list (Project SDK). After you click on the Finish button new empty Kotlin project will be created.

Now, create your first package, the steps are the same, as you would do in Java. Right-click on the "src" folder in the project structure tree **→ New → Package**. Give some meaningful name to it like:

com.example.kotlin

Finally, create your first Kotlin source code file by right-clicking on a package you just created and choosing: **New → Kotlin File / Class**. Give some name to a file, like for example: First. A newly created file will appear in your package.

Great! We have now an empty Kotlin file. We are ready to add some code!

Kotlin and Gradle

In this section, we will cover the Kotlin project setup with Gradle. To do that we need a plain Kotlin Gradle project for the demonstration purpose. Locate the

”PlainProject” folder from Fundamental Kotlin examples root directory. Directory content represents raw projects ready for importing and building.

We will import it into IntelliJ IDEA and explain each important file and directory. Open IntelliJ Idea IDE and then choose:

File → Open

locate this directory, confirm opening by clicking on the Open button.

IntelliJ Idea will show you a dialog titled “Import Project from Gradle”. Then, select the option: “Use Gradle wrapper task configuration”. By selecting this option Gradle Wrapper will be initialized.

Gradle Wrapper is recommended way to execute any Gradle build. It is a script that invokes a declared version of Gradle configured in our files, performing dependencies download (it if necessary) and executing build procedure. Because of this developers can get up and running with a Gradle project quickly without having to follow manual installation processes.

Confirm by clicking the OK button.

It will take some time for IDE to import files. The next dialog you will see is the dialog with the title: “Gradle Project Data To Import”. Select each item and confirm by clicking OK.

As you can see project consists of several important files and directories. Inside the root of the project, we see several important files. Let’s take a look at them.

settings.gradle: file containing a list of project modules that we will build:

```
include ':WelcomeToKotlin'
```

This project has one main module named “WelcomeToKotlin”. Modules represent sub-units of your main Gradle project. Thanks to this it is possible to better organize the code and achieve better separation. For example, we may have the main application module and several additional modules for various purposes, such as libraries or plugins.

.gitignore: configuration file for Git, it has the purpose of preventing versioning of trash files

build.gradle: it is located in the root directory of the project. It defines main repositories and library (or plugin) dependencies that will be used for modules and build script itself. It also defines the version of Kotlin that will be used:

```
buildscript {
    ext.kotlin_version = "1.7.10"

    repositories {

        google()
        mavenCentral()
    }

    dependencies {

        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"

    }
}

allprojects {

    repositories {

        google()
        mavenCentral()
    }

    apply plugin: "kotlin"

    dependencies {

        api "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"

    }
}
```

As its name suggests “buildscript” block defines dependencies for the build script itself. All build script dependencies will be pulled from “Google” and “Maven Central”. We will use the same repositories for build dependencies as well.

Take a look at the ”allprojects” block. This block is used to add configuration items that will be applied to all modules **and the root project as well**. Besides the ”allprojects” block we may have defined the ”subprojects” block too. This block can be used to add configuration stuff for all modules **only**. These blocks can be used as many times as we want in the root project (root ”build.gradle” configuration file).

We will apply the “kotlin” plugin and add the dependency to the standard Kotlin library. This will be enough for us to write Kotlin code in all of our modules. It is important to note that in most cases dependencies can be added with ”api” and ”implementation” directives. The difference between “api” and “implementation” directives is that if you are using “api” all other modules that depend on the parent module will inherit the dependency. In the case of “implementation”, it is required to provide your dependency for the particular library for each module.

Each Gradle module has its own ”build.gradle” configuration file that defines additional plugins that will be applied and dependencies for libraries that will be used specifically for that module. The same applies for ”build.gradle” configuration located under the “WelcomeToKotlin” directory (module):

```
apply plugin: 'application'

application {
   mainClassName =
      "net.milosvasic.fundamental.kotlin.WelcomeKt"
}

dependencies {

   // Your dependencies go here
}
```

You can see from this configuration file that our application module will use the “application” plugin with the fully-qualified main class name defined. This is the entry point of our application. We will talk more about fully qualified naming in later sections of the book.

Note:

If you are not familiar with Gradle invest some time to learn it, you will need it for more advanced project setups.

Expand the ”src” directory until you notice the ”Welcome.kt” source code file:

```
package net.milosvasic.fundamental.kotlin

fun main() {

    println("Welcome to Kotlin.")
}
```

As you can see, this is another “hello world” program. All Kotlin source code files are kept in the ”main/kotlin” directory. Java source code is usually located in the ”main/java” directory.

Right-click on the ”Welcome.kt” file and choose Run. After the program is executed you should see the following output:

```
Welcome to Kotlin.
```

You are now ready to extend this project with your code or to create a new one from scratch based on this example.

Creating Gradle project

In this section, we will show you how to create a Gradle project from zero and generate Gradle configuration files. We will demonstrate the use of the Gradle initialization wizard that will create the Gradle application project with a flat structure (no additional modules). All configuration files will be contained in one single directory – the root of the project.

Create a directory where you want to initialize your project:

```
$ mkdir my_funny_project
```

Cd into the directory:

```
$ cd my_funny_project
```

Now invoke Gradle initialization command:

```
$ gradle init
```

Note:

To be able to invoke the "gradle" command Gradle must be installed on your computer and its binaries added to your system path. For more information on how to install Gradle on your computer take a look at the official documentation at the following address:

https://gradle.org/install/

Wizard is interactive and it will ask you a couple of questions:

Select type of project to generate:
1: basic
2: application
3: library
4: Gradle plugin
Enter selection (default: basic) [1..4]

We choose option number 2.

Select implementation language:
1: C++
2: Groovy
3: Java
4: Kotlin
5: Swift
Enter selection (default: Java) [1..5]

We choose option number 4.

Select build script DSL:
1: Groovy
2: Kotlin
Enter selection (default: Kotlin) [1..2]

This one is tricky! We choose option number 1 because previous Gradle configurations files examples were written in Groovy. Groovy is the default language for writing Gradle scripts. If you like you may choose Kotlin instead of Groovy.

Project name (default: my_funny_project): ***my_funny_project***

Source package (default: my_funny_project): ***examples***

You will be asked to set the project name and the package. After that Gradle will perform the initial build:

```
BUILD SUCCESSFUL in 27s
2 actionable tasks: 2 executed
```

Your project is initialized and ready. Depending on the version of Gradle that you have installed and the DSL language that you chose in initialization wizard generated files may differ when compared to examples that we have shown previously. Take a look at them closely and chose the configuration that you more prefer. For the scope of this book, you must be now ready to write, build and run your code.

Converting Java source code to Kotlin

If you have a legacy project written in Java it is possible to convert its code into Kotlin. IntelliJ Idea offers a very powerful feature: Java to Kotlin conversion.

For example, if you paste some Java code into a .kt file IDE will ask you to perform the conversion. This is great because you do not have to rewrite the code snippet. All you have to do after conversion is to do some cleanup and adjust formatting.

Also, if you have a significant amount of Java classes you can convert them. To do code conversion, locate your Java source code (.java) file the chose:

Code -> Convert Java File to Kotlin File.

Let’s try this. Import the ”CodeExamples” directory containing book code examples Kotlin project into your “IntelliJ Idea” IDE if you did not already. Locate the “ConvertMePlease.java” file from Fundamental Kotlin examples:

```
public class ConvertMePlease {

    private String myFieldToBEConverted;

    public ConvertMePlease(String myFieldToBEConverted) {

        this.myFieldToBEConverted = myFieldToBEConverted;
    }
}
```

Then right-click to file, choose **Code -> Convert Java File to Kotlin File**.

Take a look at the code you have now:

```
class ConvertMePlease(

    private val myFieldToBEConverted: String
)
```

Note:

You may be asked by IDE to accept corrections (meaning removing the .java file and putting .kt file in its place). Feel free to respond positively to this.

As you can see the code is not just converted, but that the amount of the code is significantly reduced. This is one of the great treats that Kotlin gives to you! We will talk about that in later sections of this book.

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/

- Python: https://www.python.org/
- C#: https://docs.microsoft.com/en-us/dotnet/csharp/
- Gosu: https://gosu-lang.github.io/
- ML: https://en.wikipedia.org/wiki/ML_(programming_language).

The lexical structure of Kotlin programs

The lexical structure represents a set of basic rules that define how you write programs. In this section, we will explain the lexical structure of the Kotlin programming language.

We will cover:

- Unicode character set
- Case sensitivity
- Whitespace
- Comments
- Identifiers
- Literals
- Reserved words.

Unicode character set

Unicode represents a standard for the consistent encoding, representation, and handling of text. Unicode is expressed in most writing systems. Kotlin programs are written using Unicode. In Kotlin you can use Unicode characters everywhere. For example for giving names to your constants and variables, for comments, and so on. What makes Unicode so powerful is that it can represent virtually every written language in common use on the planet! To illustrate to you that this can be done, we have created one simple example. Open “unicode_example.kt” from the book’s code examples and take a look at them:

```
package net.milosvasic.fundamental.kotlin.lexical_structure

class МојаКласаНаЋирилици {

    val поздрав = "Поздрав на ћирилици!"
}

fun main() {

    val unicodeClass = МојаКласаНаЋирилици()
    println(unicodeClass.поздрав)
}
```

This example uses Serbian Cyrillic for the class name, for the name of the class field, and the value of the field itself. If you run this program, the following output will be printed out:

```
Поздрав на ћирилици!
```

Even though it is possible to write your code in Serbian Cyrillic or in Chinese if you want, it is recommended to use only ASCII characters for this purpose. In the case of this code proper warning will be seen (check this out in your IDE): “Non-ASCII characters in an identifier”.

Note:

ASCII is a character encoding standard. ASCII codes are used for representing textual content in computers and other devices. ASCII stands for American Standard Code for Information Interchange and consists of (only) 128 characters.

Case sensitivity

Kotlin is case sensitive programming language. That means that if you do something like this:

```
val myConstant = 0
val myCONSTANT = 1
val MYconstant = 2

fun main() {

    println("myConstant = $myConstant")
    println("myCONSTANT = $myCONSTANT")
    println("MYconstant = $MYconstant")
}
```

You have defined three different constants. The output of this small program is the following:

```
myConstant = 0
myCONSTANT = 1
```

```
MYconstant = 2
```

Based on this, in Kotlin “While” and “while” are not the same thing! First may be used as a name for the class, or constant, or for the name of the variable. Later one is reserved keyword by Kotlin programming language. It cannot be used for the naming of classes, constants, or variables.

Whitespace

Kotlin ignores whitespace. Spaces, tabs, newlines, and other whitespace are ignored except when it appears within quoted characters and string literals.

Comments

Comments are human-readable explanations in the source code. They are added to make the source code easier for developers to understand. Comments are ignored by the Kotlin compiler. In the case of Kotlin, there are four types of comments that programmers can write.

Single line comments: begin with the characters “//” and continue until the end of the current line.

```
// Single-line comment
```

Multi-line comments: begin with the characters “/*” and continue, over any number of lines, until the characters “*/” define its ending. Any text between the opening and closing characters is ignored by the Kotlin compiler.

```
/* Multi-line
* comment */
```

Source code documentation comments: begin with “/**”, for documentation **multi-line** opening and finish with closing characters: “*/”. This type of comment is regarded as a special documentation comment. When you write a Kotlin class or function that will be delivered to other developers, providing proper documentation comments is required so all information is there, such as the purpose of the class (or function), parameters, or return data information.

```
/**
 * Documenting source code.
 *
 * @param file The file to upload.
 * @return Result of the upload.
 * @author Milos Vasic
 */
```

To generate documentation from source code in some format like HTML developers use documentation engines. For Kotlin Dokka documentation engine can be used:

https://github.com/Kotlin/dokka

Dokka is a documentation engine for Kotlin and it fully supports mixed-use of languages (Java and Kotlin) in parallel. Dokka understands standard Javadoc source code documentation comments for Java source code files and KDoc comments in Kotlin source code files. Multiple output formats are supported:

- Standard Javadoc,
- HTML
- Markdown

Nested multi-line comments: Unlike Java, Kotlin supports nested multi-line comments. Thanks to this you can comment out large blocks of code quickly and easily. Even if the code already contains multiline comments!

```
/*
*   /*
*       /*
*           // Nested multi-line comment!
*       */
*   */
*/
```

Identifiers

Identifiers are the names given to constants, variables, classes, functions, and so on. For example:

```
val index = 101.1
```

In this example,"val" is a keyword, and "index" is the name (identifier) given to it.

There are certain rules and conventions for naming in Kotlin that we must follow or the Kotlin compiler will complain:

- Identifier starts with a letter or underscores followed by zero, letter, and digits
- Whitespaces are not allowed
- An identifier cannot contain special symbols, such as “@”, “#”, “!” and so on
- As we already mentioned, identifiers are case sensitive

Let’s take a look at the list of some valid identifiers:

- book
- myClass
- doorsNo1

Opposite to this list, we have a list of invalid ones:

- class
- while
- 1doorNo
- identifier with some spacing
- @!#name

The first two items are invalid because these words are reserved by the Kotlin programming language. We will talk more about this soon. The third one starts with a number, which is not allowed. As we already mentioned, whitespaces are not allowed for identifiers so the fourth item is invalid too. The last one contains special characters.

Literals

Literals are sequences of source code characters that directly represent constant values. Literals appear as is in Kotlin source code. They include integers, floating-point numbers, single characters within single quotes, strings of characters within double-quotes. Also reserved ”true” and ”false” words for booleans and null. Let’s take a look at some literals:

```
100
100.0
'c'
100L
"hundred"
true
false
null
```

Reserved words

All words used by the Kotlin programming language consider being “reserved”. That means that these words cannot be used as identifiers of your classes, constants, variables, or functions. This is the list of reserved words in the current version of the Kotlin:

package
as
typealias
class
this
super
val
var
fun
for
null
true
false
is
in
throw
return
break
continue
object
if
try
else
while
do
when
interface
typeof

Packages and code organization

You probably remember our first example: “First.kt”. The first line of the code in that example defines belonging to a package:

package net.milosvasic.fundamental.kotlin

A package is a group of similar types of classes, interfaces, and sub-packages. Packages represent the basic level of organization of our source code. Every ".kt" file must belong to some package. We should define our package at the top of our ".kt" files.

The structure of directories containing this is the following:

```
net
  milosvasic
    fundamental
      kotlin
```

Note:

It is not required to match directories and packages. Placing the source files in the filesystem can be arbitrary.

Naming conventions for package names are the following:

- to avoid conflicts with the names of classes, objects, or interfaces all letters are written in lower case letters
- root package is usually reversed internet domain name, like for example:

com.google

- in some cases, reversed internet domain name may not be a valid package name This can happen if the domain name contains a hyphen or other special character or if the package name begins with a digit or other character that is illegal to use or if the package name contains a reserved keyword. In such situations, it is recommended to use the underscore character: "_".

Let's have a look at an example of the organization of the code. Let's assume that we have a simple notes application that can create, edit and delete notes. The main package where all code resides could be called:

com.notes

Then, inside this package we will expand our organization hierarchy to support various code responsibilities:

com.notes.models
com.notes.models.actions
*com.notes.*connectivity
com.notes.utils

and so on.

Note:

If the package is not specified in the source code file, the contents of such a file belong to the default package that has no name.

As you can see, it is not difficult to organize source code. As you gain more experience with Kotlin programming you will be more clever in making decisions where to put your source code.

Importing source code

Source code is imported by using the “import” keyword. “import” is used along with importing name (scope) passed to it. We can import a single name or all the accessible contents of scope, such as:

- package
- class
- object

and so on.

In Kotlin certain number of packages is imported by default:

- kotlin.*
- kotlin.annotation.*
- kotlin.collections.*
- kotlin.comparisons.*
- kotlin.io.*
- kotlin.ranges.*
- kotlin.sequences.*
- kotlin.text.*

Depending on the targeting platform there are a few additional packages that are imported. If your project is targeting JVM:

- java.lang.*
- kotlin.jvm.*

Or, if your projects are targeting JavaScript:

- kotlin.js.*

Let’s have a look at some ”import” examples:

```
// User is now accessible without qualification:
import com.example.User

// Everything in 'com.example' becomes accessible:
import com.example.*
```

If there is a name clash, we can rely on “as” keyword. “as” has a purpose of locally renaming the clashing entity:

```
// User is accessible:
import com.example.User

// TestUser represents now: “org.tester.User”
import org.tester.User as TestUser
```

“import” can be also used to import other declarations, such as:

- enum constants
- functions and properties declared in object declarations
- top-level functions and properties.

Note:

If a top-level declaration for the imported entity is marked as ”private”, then it means that it is ”private” to the file in which it is declared. We cannot import “private” entities. We will talk more about “private” and other “visibility modifiers” in upcoming sections of this book.

The statements and expressions

What is a statement? Statement represents a syntactic unit of programming language that expresses some action to be performed. They are everything that makes up a complete unit of execution. We will start with very basic statements in this section and progress towards more complex examples.

Let’s have a look at few statements:

```
val result = 5 * 5
```

In this example, we have defined a statement in which multiplication operation is resolved into the value that is assigned to a variable.

Any valid unit of code that resolves to a value represents expression. Expressions are part of statements.

```
println(“Hello!”)

var x = 5

val parameter = if (x > y) 10 else 100
```

Al these are statements.

Grouping multiple statements (none or more) between braces “{ }” is called a block of code. Blocks represent basic scopes in which values are defined and to which we can access. We will talk more about scopes and values in the next section.

Let’s make some code blocks (“Blocks.kt” source code file):

```
fun main(args: Array<String>) { // <- Main function block
                                // start

    if (args.size == 2) { // <-Start of 'if' block

        // Statements:
        print("1st argument: ${args[0]}")
        print("2nd argument: ${args[0]}")

    } // <- End of 'if' block

} // <- Main function block end
```

Constants and variables

Constants and variables are the names that you give to computer memory locations that are used to store the values of a computer program. Constants do not change their value over time. A variable, on the other hand, changes its value. In Kotlin we can define a variable and constant values both.

Each constant or variable belongs to its parent scope. The scope of a variable (or constant) is the region of a program in which that variable (or constant) is visible. In parent scope variables (or constants) are accessible by their name and can be used. If we try to access variables outside of their scope compiler will complain and report an error.

Variables (and constants) can be local or global. A local variable is a variable that is given local scope. Local variables reference in the function or a block in which it is declared. Global variables are declared outside any function, and they can be accessed (used) on any function in the program.

Let's have a look at some examples. Open ”Values.kt” source code file.

The first example in this section is an example of defining a constant. We will assign the value that can be only read:

```
// We provide the type and the value:
val x: Int = 1
```

This statement consists of several parts. Reserved word “val” tells the compiler that we will define a constant, which is then followed by the constant’s name. In our case name of this constant is “x”. The data type of the constant is specified after the constant’s name with “:” character followed by the data type. In this case, it is a number (Integer). Finally, we assign the value like in math: “= 1”. We have defined a constant named “x” with the value of ”1” (which is an Integer data type).

Value for x cannot be reassigned. We can make this statement simpler because the Integer type can be inferred from the value that we are passing to a constant. Let’s define another constant:

```
val y = 1
```

“y” constant follows the same rule. As you can see, since type is inferred we do not have to mention it explicitly.

What will happen if we do not provide the type or we do not provide value for the constant? The following line will produce an error:

```
val z: Int
```

And finally, to define variable instead of constant (to be mutable), we will use the “var” word instead of “val”:

```
var m = 1
```

“m” variable can change its value after it has been initialized:

```
var m = 1
println("M is: $m")
m = 2
println("M is: $m")
m = 3
println("M is: $m")
```

The output of the code snippet will be the following:

```
M is: 1
M is: 2
M is: 3
```

As you just saw, defining constants and variables is simple. Let’s put the scope into play (“Scope.kt”):

```
fun main() {

   val global = "Global"

   fun myFunction() {

       val local = "Local"

       // We can access to global scope:
       println("Local: $local")
       println("Global: $global")
   }
```

```
    // We can’t access to local scope,
    // compiler would complain:
    // println("Local: $local")
    println("Global: $global")
}
```

This example illustrates local vs global scope for the values. Inside the program’s main function we have defined a constant named “global”. Inside the “myFunction” function we have defined a constant named “local”. “myFunction” function can access variables and constants of the outer scope. However, the outer scope cannot access to “myFunction” function’s local scope.

Working with functions

Function (or method) is a block of reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing. The function is a named (and callable) piece of code that can accept zero or more arguments that will be used inside the function’s block and can (but don’t have to) return a value as a result of its work.

Functions in Kotlin are declared using the reserved word ”fun”:

```
val result = execute(25)
```

In this example, we have called a function with the name “execute” that accepts one argument of Integer type and returns a result. The result of function execution is assigned to a constant named “result”.

Let’s see some functions through examples. Open the file with the name ”Functions.kt” from the book’s code examples.

This function receives two numbers (Integers) as parameters and returns the sum of numbers:

```
fun sum(x: Int, y: Int): Int {

    return x + y
}
```

We can simplify it. Take a look at new version of that function:

```
fun sumSimplified(x: Int, y: Int) = x + y
```

We introduced a function with an expression body and inferred return type.

In the same file we have a function that does not return a value, for that purpose we use Unit type:

```
fun printSum(x: Int, y: Int): Unit {

   println("Sum is ${x + y}")
}
```

Since our function does not return anything, we can leave out the Unit return type:

```
fun printSumSimplified(x: Int, y: Int) {

   println("Sum is ${x + y}")
}
```

Passing arguments to functions

In this section, we will show how to pass arguments to functions. We will present a couple of simple examples of this. Open “PassingArguments.kt” from the book code examples:

```
// Function accepts 2 Integer arguments and returns
// its multiplication:
fun multiply(a: Int, b: Int) = a * b

// Accepts 2 Integer arguments and returns its division:
fun divide(a: Int, b: Int) = a / b

// Main program function receives arguments from command line:
fun main(args: Array<String>) {

   val x = 5
   val y = 7

   // We are passing values of 'x' and 'y' constants as
   // parameters to 'multiply' function:
   val z = multiply(x, y)
```

```
    // We are passing result(s) of 'sum' function as arguments
    // to 'divide' function:
    val m = divide(
            sum(x, y),
            sum(1, 2)
    )

    // We are passing String (a word) as a parameter to
    // 'println' function:
    println("Z: $z, M: $m")
}
```

The output of the program will be:

```
Z: 35, M: 4
```

Follow the order and results of functions execution to verify the final result (value) of ‘z’ and ‘m’ constants.

Default arguments

In Kotlin we can define default values for function arguments (parameters). That means that if we do not pass value for the argument, default value will be used. Let’s see how it works in the following example (“DefaultArguments.kt” from book code examples):

```
    fun calculate(
        a: Int, b: Int = 10, c: Int = 20, d: Int = 30
    ): Int {

        return a + b + c
    }

    // We will use all default values for parameters
    // (passing value for 'a' is mandatory):
    calculate(10)

    // We will pass value for every argument of the function:
    calculate(1, 2, 3, 4)

    // We will pass values for 'a' and 'c' arguments,
    // all others will use its default values:
    calculate(100, c = 100)
```

We have defined function with name ‘calculate’. Function accepts four arguments. First argument (“a”) does not have default value. However, the rest of arguments have default values: 10, 20 and 30 for “b”, “c” and “d”.

As you can see in code comments, we may have various combinations in function use. In last call of the function we have passed value to the argument by its name. This is called “named argument”. We will talk more about named arguments in upcoming sections of this book.

Working with exceptions

Exception represents the problem that occurs during the execution of a program. When an Exception occurs the normal flow of the program is disrupted and our program terminates abnormally. To avoid program termination exceptions must be handled. The exception can happen for many various reasons.

Some exceptions are caused by user error, some others are caused by programmer error, while the others by physical resources that have failed in some manner (filesystem access, network, etc).

From book code examples open “Exceptions.kt”:

```
@Throws(IllegalArgumentException::class)
fun getCarPrice(model: String) = when (model) {

    "Mercedes" -> 100
    "BMW" -> 200
    "Opel" -> 300
    else -> throw IllegalArgumentException(
      "We do not recognize '$model' car model"
    )
}

val models = listOf("Mercedes", "Opel", "Fiat")

models.forEach {
    try {
        val price = getCarPrice(it)
        println("Price of '$it' is $price")
    } catch (e: IllegalArgumentException) {

        println(e.message)
    }
}
```

Let’s explain the code from above. “getCarPrice” function returns the price of a car based on a model name. If we do not recognize provided model we are throwing “IllegalArgumentException”. “IllegalArgumentException” is thrown to indicate that a function has been passed an illegal or inappropriate argument. Besides this, a frequently used exception is “IllegalStateException” to indicate that a function has been invoked at an illegal or inappropriate time. In other words, our program is not in an appropriate state for the requested operation.

In this program, we are printing prices for each car model from the “models” collection. To prevent the program from crashing, we are catching an exception that may be thrown by “getCarPrice” function.

Let’s run our example:

```
Price of 'Mercedes' is 100
Price of 'Opel' is 300
We do not recognize 'Fiat' car model
```

Note:

Unlike Java, Kotlin does not support checked exceptions (exceptions that are notified by the compiler at compilation time).

Throwable

In this section, we will focus on the “Throwable” data type (class) and explain some hierarchy that exists behind this kind of data type (class). In Kotlin the “Throwable” class is the parent class for all “Exception” classes. Every exception has a message, stack trace (a list of frames that starts at the current function and extends to when the program started), and an optional cause. Besides than ”Exception” class, there is another subclass (data type): “Error”. “Error” is derived from the “Throwable” class too.

Errors represent the abnormal conditions that can happen in case of severe failures. **They are not handled by programs!** Errors are generated to indicate errors generated by the runtime environment. For example, when JVM reaches the point when it is out of memory. Programs cannot recover from this type of errors.

“Exception” data type (class) has two main sub-types (subclasses):

- “IOException” class: signals that I/O exception of some sort has occurred. It represents the general class of exceptions produced by faulty or interrupted I/O operations.

- ”RuntimeException” class: represents a superclass of exceptions that can be thrown during the regular operation of the JVM. ”RuntimeException” and its subclasses are unchecked exceptions. Unchecked exceptions do not need to be declared in a function or constructor's throws clause if they can be thrown by the execution of the function or constructor and propagate outside the function or constructor boundary.

Note:

We will talk about classes and subclassing (inheritance) in upcoming sections of this book.

From book code examples open “Throwable.kt”:

```
fun fail(t: Throwable) {

    println(t.message)
    exitProcess(1)
}

@Throws(Throwable::class)
fun process(what: Int) {

    if (what < 0) {

        throw Throwable("Error")
    }
    println("Processing: $what")
}
```

```
val items = listOf(2, 4, 6, 0, -2, -4, -6)
items.forEach {
    try {

        process(it)
    } catch (t: Throwable) {

        fail(t)
    }
}
```

This example illustrates the use of the "Throwable" data type. As you can see, the way we use it and the way we handle it is very similar to the way that we use the "Exceptions" data type. Executing our program will produce the following output:

```
Processing: 2
Processing: 4
Processing: 6
Processing: 0
Error
```

Note:

The throw is an expression in Kotlin, so you can use it, for example, as part of an Elvis expression:

```
val model = car.model ?: throw IllegalArgumentException(

    "Model is mandatory"
)
```

Try / Catch / Finally block

The "try/catch/finally" block helps in writing the program code which may throw exceptions in its runtime. By using it gives us a chance to recover from exceptions by executing alternate program logic or handle the exception gracefully. Most important, it helps in preventing ugly application crashes.

Comparing to Java, there is only one difference: **a try expression is truly an expression in Kotlin.** That practically means that you can assign its result to a variable.

From book code examples open ”TryCatchFinally.kt”:

```
@Throws(IllegalArgumentException::class)
fun process(what: Int) {

    if (what < 0) {
        throw IllegalArgumentException(
            "Invalid parameter: $what"
        )
    }
    println("Processing: $what")
}

val items = listOf(2, 4, 6, 0, -2, -4, -6)
items.forEach {

    println("Processing started")
    try {

        process(it)
    } catch (t: IllegalArgumentException) {

        println(t.message)
    } finally {

        println("Processing completed")
    }
}
```

Pay attention to the “try/catch/finally” block. If we execute this program the following output will be produced:

```
Processing started
Processing: 2
Processing completed

Processing started
Processing: 4
Processing completed

Processing started
```

```
Processing: 6
Processing completed

Processing started
Processing: 0
Processing completed

Processing started
Invalid parameter: -2
Processing completed

Processing started
Invalid parameter: -4
Processing completed

Processing started
Invalid parameter: -6
Processing completed
```

Let’s have a look at example of “try / catch” used as expression:

```
@Throws(IllegalArgumentException::class)
fun getCarPrice(model: String) = when (model) {

    "Mercedes" -> 100
    "BMW" -> 200
    "Opel" -> 300
    else -> throw IllegalArgumentException(
        "We do not recognize '$model' car model"
    )
}

val models = listOf("Fiat", "BMW", "Opel", "Audi")
models.forEach {

    println("Getting price for: '$it'")

    val price = try {

        getCarPrice(it)
    } catch (e: IllegalArgumentException) {
        0
    }

    if (price > 0) {

        println("Price obtained: $price")
    } else {

        println("Price is not available")
    }
    println("- - -")
}
```

As you can see constant “price” will receive the returned value of the “getCarPrice” function unless it throws an exception. If that happens, after we catch the exception, zero is set as the value. Executing this code snippet will produce the following output:

```
Getting price for: 'Fiat'
Price is not available
- - -
Getting price for: 'BMW'
Price obtained: 200
- - -
Getting price for: 'Opel'
Price obtained: 300
- - -
Getting price for: 'Audi'
Price is not available
- - -
```

The most frequently used exception types

In this section, we will highlight some of the most commonly used exceptions and explain their purposes.

Some of the most commonly used exceptions are:

- IllegalArgumentException
- IllegalStateException

- NumberFormatException: thrown to indicate that the application has attempted to convert a string to one of the numeric types, but that the string does not have the appropriate format

- RuntimeException: represents the superclass of those exceptions that can be thrown during the normal operation of the JVM. ”RuntimeException” and its subclasses are unchecked exceptions. Unchecked exceptions do not need to be declared in a function or constructor's throws clause if they can be thrown by the execution of the function or constructor and propagate outside the function or constructor boundary.

- NoSuchMethodException: thrown when a particular function (method) cannot be found.

- ClassCastException: thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance.

- ParseException: informs that an error has been reached unexpectedly while parsing is performed.

Data type fundamentals

In upcoming sections, we will be covering fundamental data types that you can use in Kotlin to express information. In everyday development you will be using number data types, data types for presenting textual contents, types for handling logical states (booleans), arrays, and data with “no data”, better known as “null”.

Basic data types are the foundation for building more complex things and representation of all your application data. Therefore it is really important to have a good understanding of each of them and to know when to use each of these types.

Numbers in Kotlin

We will start this section by presenting several literals that Kotlin supports:

```
1
100
5000000000
10L
```

```
-3
-3.3
0.5
1.1f
0x0F
0b01001001
```

Each of these belongs to a particular Kotlin number class.

```
1
100
-3
```

Are decimal numbers, represented with Kotlin’s “Int” (integer) class. Memory consumption by an integer is 32 bits. Minimal value is: -2,147,483,648. Maximal value is: 2,147,483,647.

```
5000000000
10L
```

Are ”Long” decimal numbers with memory consumption of 64 bits. Minimal value is: -9,223,372,036,854,775,808. Maximal value is: 9,223,372,036,854,775,807.

Kotlin also supports ”Short” and ”Byte” data types. ”Short” numbers consume 16 bits in memory with a minimal value: -32768 and a maximal value: 32767. “Byte” type consumes only 8 bits with a minimal value of -128 and maximum: 127.

```
-3.3
0.5
```

Are floating point numbers represented by the “Double” Kotlin class. It is the default class type for floating-point numbers. It consumes 64 bits of memory.

```
1.1f
```

Is an example of a floating-point that consumes 32 bits and it is represented with 32 bits in the memory. Notice the “f” suffix at the end of the literal. It tells Kotlin compiler that we want a 32-bit version of a floating-point number, that is “Float”.

```
0x0F
```

Is an example of a hexadecimal number literal.

```
0b01001001
```

Is an example of binary literal.

Note:

Kotlin does no support octal literals.

In Kotlin you can use underscores to achieve numbers readability. Take a look at the “underscoredNumbers” function inside the “Numbers.kt” file from the book’s code examples:

```
fun underscoredNumbers() {

    val millions = 5_000_000
    val hexBytes = 0xFF_EE_FF_EE
    val bigOne = 1111_2222_3333_4444L
    val bytes = 0b11001100_11001100_11001100_11001100
}
```

Check out “numbers()” function too:

```
fun numbers() {

    val a = 1 // "Int" number type
    val b = a.toLong() // Convert "Int" into "Long"

    println("a: $a") // prints: 1
    println("b: $b") // prints: 1
    print(a == b.toInt()) // We compare "a" and "b",
                          // it prints: true,
                          // which is Boolean data type.
}
```

As you can see we have defined two constants with different data types. Constant “a” is a simple Integer (“Int” class) with the value of 1. Constant “b” is “Long” but it is initialized with the value of constant “a”. In this example, we performed two conversions. We have converted “a” which type is “Int” into

“Long” by calling the “toLong()” function, and we converted “b” into “Int” by calling the “toInt()” function.

So, the conversation between numbers is done by one of the functions (each Kotlin number class has them):

- “toByte()” converts into “Byte” data type
- “toShort()” converts into “Short” data type
- “toInt()” converts into “Int” data type
- “toLong()” converts into “Long” data type
- “toFloat()” converts into “Float” data type
- “toDouble()” converts into “Double” data type
- “toChar()” converts into “Char data type

The type is inferred from the context. Also, the arithmetical operations are overloaded for appropriate conversions. For example:

```
val x = 1L + 1 // Long + Int = Long
```

All arithmetical operations are defined as operators defined in number classes. A standard set of arithmetical operations is supported, such as “+”, “-”, “/”, “*” etc.

“Int” and”Long” classes also support bitwise operations provided with proper functions:

- “shl(bits)”: signed shift left
- “shr(bits)”: signed shift right
- “ushr(bits)”: unsigned shift right
- “and(bits)”: bitwise and
- “or(bits)”: bitwise or
- “xor(bits)”: bitwise xor
- “inv()”: bitwise inversion

Characters in Kotlin

The ”Char” data type in Kotlin represents Unicode characters. Characters can be converted into numbers. To convert them use “toInt()” function. Each literal character goes in single quotes, for example: ‘1’, ‘m’, ‘p’, and so on. Also, there are some important escape sequences that we will need: \t , \b , \n , \r , \' , \" , \\ and \$. Let’s take a look at the “Characters.kt” example:

```
fun characters() {

   val tab = '\t'
   val quotes = '\"'
   val dollar = '\$'
   val character = 'c'
   print("$character$tab$dollar$tab$quotes")
}
```

Executing this example would give you the following output:

```
c	$	"
```

“c” character, dollar, and quotes are printed separated with tab.

Booleans in Kotlin

Class “Boolean” represents boolean data type. “Boolean” has two possible values: ”true” or ”false” which is the base of logic in computer programming.

There are 3 major operations we can perform on booleans in Kotlin:

“||”, lazy disjunction, for example, a || b
“&&”, lazy conjunction, for example, a && b
“!”, negation, for example: !a || !b

Open “Booleans.kt” from code examples of the book:

```
fun booleans() {

   val number = 1
   val a = number == 1
   val b = number == 2
   println("a || b -> ${a || b}")
   println("a && b -> ${a && b}")
   println("!a || !b -> ${!a || !b}")
}
```

Running this example will give you the following output:

```
a || b -> true
a && b -> false
```

```
!a || !b -> true
```

As you will see in the upcoming section of the book, booleans are the foundation of any program's business logic.

Arrays

Arrays represent data structures that consist of a collection of elements. Each element is identified by at least one array index or key. To understand arrays in Kotlin open "Arrays.kt" and take a look at simple self-explanatory code. Let's go through examples.

The first example illustrates the creation of an array that contains five integer numbers:

```
val myArray = arrayOf(2, 4, 6, 8, 10)
```

As you can see "arrayOf" function crates array that contains members that are all parameters that we have passed to the function. The name of this array is "myArray".

To create array of 10 members which values are generated by: member index + 100 you can achieve like this:

```
val a = Array(5, { i -> i + 100 })
```

In this example, we have created an array named "a" with members: "100, 101, 102, 103, 104". This example calls Array's class constructor that accepts two parameters: number of items in the array, lambda function that creates values for each array member based on the current index. We will talk more about class constructors and lambda functions in the upcoming parts of this book.

To print all of "a" array members you can use the following function:

```
fun printArray(array: Array<*>) {

   array.forEachIndexed { index, value ->
      println("$index -> $value")
   }
}
```

Running this function will give the following output:

```
0 -> 100
1 -> 101
2 -> 102
3 -> 103
4 -> 104
```

To set the value to 1000 to the element at a position at index 2 do the following:

```
a[2] = 1000
```

As you can see, we have access to the third element of “a” array and set a new value. The first element of the array is at the index of 0.

Accessing the first member of the array and assigning its value to a constant (named “y”) can be done like this:

```
val y = a[0]
```

“y” has now a value of “100”.

Array can have members of various types:

```
val b = arrayOf(2, 4, 6, "Some string", "One more string", 8, 10)
```

Or to have all “nulls” for the value of its members:

```
val c = arrayOfNulls<Int>(5)
```

In this example, the “arrayOfNulls” function creates an array with five null elements. The expected type of members (once we assign something to them) is ”Int” (integer). We will talk about the null type soon. For now, it is enough that you understand that all five members of this array do not have values assigned.

And finally, an array of integers can be created like this too:

```
val d = intArrayOf(10, 100, 1000, 10000)
```

However, this is not the same as:

```
val e = arrayOf(10, 100, 1000, 10000)
```

Since “d” has the type of “IntArray”, while “e” has the type of “Array”. For arrays of other types, there are functions like “charArrayOf()”, “longArrayOf()”, and so on. Or, you can just use the “arrayOf()” function for everything.

Strings in Kotlin

String data type represents a sequence of characters. In Kotlin, Strings are immutable. Elements in one String are characters that can be accessed like any other array.

Let’s have a look at the ”Strings.kt” source code file. We will start with defining some simple strings:

```
val s1 = "Some string"
```

In this example, we have defined a constant named “s1” with the value of: “Some string”.

Then, the next example is with the escaped character. In this example we are escaping tab character:

```
val s2 = "Some\tstring\n"
```

Printing the value of “s2” gives the following output:

```
Some    string
```

Kotlin supports the definition of multi-line strings too:

Multiline string:

```
val s3 = """
        Raw string example ...
        We have multiline here!
    """
```

Note:

Multiline strings preserve all white spacing we define inside the string. To avoid that you can use the “trimIndent” function:

```
val s4 = """
```

```
        Raw string example ...
        We have multiline here!
""".trimIndent()
```

In the case of margin that is formed with the beginning of each line you can use the “trimMargin” function:

```
val s5 = """
        |Raw string example ...
        |We have multiline here!
""".trimMargin()
```

Printing both (“s4” and “s5”) constants will give us the same output:

```
Raw string example ...
We have multiline here!
```

Since we already mentioned that you can do with strings all that you do with any array, let’s print each letter of one simple word:

```
fun stringIsArray(word: String) {

    word.forEach {
        println(it)
    }
}

fun main() = stringIsArray("Elephant")
```

“stringIsArray” function will print each character of the word “Elephant”:

```
E
l
e
p
h
a
n
t
```

We will work more with strings in upcoming sections of this book.

String templates

Let’s go back to a file Second.kt and the function called printSum. Take a look at the following:

```
println("Sum is ${x + y}")
```

You will notice that we used the “${ … }” string template expression. We reduced the code boilerplate we would have in Java. We avoided String concatenation with + operator.

The following examples will show how powerful string templates are in Kotlin. Open “StringTemplates.kt” and take a look carefully into string templates usage:

```
val firstName = "John"
val lastName = "Snow"
val title = "We called $firstName $lastName to come."
```

As you can see, it is really easy to inject values into the text. After $ sign put the name of the value that you are interested in.

```
val profession = "captain"
val subtitle = "He is: ${if (firstName == "John") profession else "sailor"}."
```

Any expression can be used.

Kotlin supports multiline text. There is no need for escaping when using multiline text. If you plan to write some regex patterns it may be very useful to you. You don't need to escape a backslash by a backslash too!

Take a look at the example of multiline text. String template entries may be used here as in previous examples:

```
val bornDay = 6
val bornYear = 1580
val bornMonth = "June"

val diedDay = 21
val diedYear = 1631
val diedMonth = "June"

val body = """
```

```
${profession.capitalize()} $firstName $lastName
($bornDay $bornMonth $bornYear - $diedDay $diedMonth $diedYear),
Admiral of New England, was an English soldier, explorer, and author.
He was knighted for his services to Sigismund Báthory, Prince of Transylvania, and
his friend Mózes Székely.
His books and maps were important in encouraging and supporting English
colonization of the New World.
He gave the name New England to the region and noted:
"Here every man may be master and owner of his own labour and land...
If he have nothing but his hands, he may...by industries quickly grow rich."
"""
```

Running “StringTemplates.kt” from the book code examples will give us the following output:

```
We called John Snow to come.
He is: captain.
Captain John Snow
(6 June 1580 - 21 June 1631),
Admiral of New England, was an English soldier, explorer, and author.
He was knighted for his services to Sigismund Báthory, Prince of Transylvania, and
his friend Mózes Székely.
His books and maps were important in encouraging and supporting English
colonization of the New World.
He gave the name New England to the region and noted:
"Here every man may be master and owner of his own labour and land...
If he have nothing but his hands, he may...by industries quickly grow rich."
```

Nullability

We have already mentioned “null” in previous sections of this book. Now we will explain its purpose. Every variable or constant that we create should point to a certain class instance. Before we go any further we will explain shortly the meaning of the “class” and the “instance”. A ”class” is the blueprint (definition) which you use to create objects in memory. An object is an instance of a class. “object” and the “instance” are the same thing, but the word “instance” indicates the relationship of an object to its class. To better illustrate this let’s use a real-world example. For example “wolf” is a ”class” of animals. But, in a herd, every wolf (member) of the herd is one instance of that class.

Variables (or constants) that do not point to anything yet have default ”null” type. Using constants and variables that do not point to any particular instance can be dangerous. In Java, it is really easy to have variables (or constants) that have a null value. Let’s have a look at the example (“NullExamples.java” from the book’s code examples):

```
String nullStringVariable;
Boolean b;
```

Both lines of code produce variables with null values. However, once we initialize these variables they do not have null value anymore:

```
nullStringVariable = “My string”;
b = true;
```

To make these variables null again we will assign null to them again:

```
nullStringVariable = null;
b = null;
```

Trying to invoke function on variable that has null value will produce ”NullPointerException”. The following code will crash your application:

```
nullStringVariable = null;
int size = nullStringVariable.length();
```

Calling the “length()” function will crash the application since “nullStringVariable” is null and does not point to any particular instance of the class.

In Kotlin working with nulls is safe. Kotlin's type system protects us from getting null pointer exceptions.

To get a null pointer exception you must do one of the following:

- Explicitly throw NullPointerException
- Use !! operator (we will talk more about this later) with null variable (or constant)
- Use Java code that is not guarded against it
- Inconsistence with initialization

In the following example, we are returning a null value by intention. Open ”Nulls.kt”:

```
/**
 * This example returns incremented value
 * if passed parameter is positive number
 * otherwise it returns null.
 */
fun incrementPositive(x: Int): Int? {
    return if (x > 0) x + 1 else null
}
```

The most important part of this example lies in the **“Int?”** part where we defined our function return type. This means, the returned value type is an integer but it may be also null. If we have defined our function return type as “Int” instead of “Int?” we could not return null.

“Int?” type is ”optional” type. Every class has its “optional” equivalent. Optional is a container class used to contain null or non-null objects. The optional object is used to represent null with an absent value. Thanks to facilities that are exposed to the user’s access to nullable data are safe and guarded.

Look into the next example:

```
/**
 * s1 - cannot assign null values!
 * s2 - can assign null values
 */
var s1: String = "this variable cannot store null references"
var s2: String? = null
```

We control whether or not we accept nulls!

Let’s consider the following scenario: what if we have or maybe we do not have null assigned to our variable (or constant) and we are not sure about it? Well, in that case, we will deal with it like in the last example from “Nulls.kt”:

```
/**
 * If word is not null print it.
 */
fun printer(word: String?) {
    word?.let {
        val uppercase = word.toUpperCase()
        println("Word [ $uppercase ]")
    }
}
```

The parameter of the function can be null and the expected type of the parameter is a string, then the type of parameter itself is “String?”, an ”optional”. To access it safely we will use the ”elvis” (“?:”) operator. Use the ”elvis” operator to access instance’s members. If the instance is null nothing will happen. We will not enter “let” code block and execution will not be performed.

“Let” is a scoping function that lets you declare a variable for a given scope. There is the most common use of ”let” when applied to a nullable reference like in our last example. The “?:” (“elvis”) operator lets you make sure that the code in scope is only executed if the expression is not null. If we are dealing with variables or function return values that are not “nullable”, ”elvis” operator is not needed:

```
/**
 * Word is not null!
 */
fun printer2(word: String) {

        val uppercase = word.toUpperCase()
        println("Word [ $uppercase ]")
}
```

As you can see, we do not need the “elvis” operator nor the “let” scoping function here because the “word” function parameter cannot be null. “word” is pure String, not an ”optional”.

Let’s rewrite our Java code from the previous section (“NullExamplesKt.kt”) into the Kotlin:

```
var b: Boolean? = null
var nullStringVariable: String? = null
```

In this code snippet we have defined two null variables. Now, let’s crash the app:

```
// Calling 'length()' function will crash the application:
val size = nullStringVariable!!.length
println("Length is: $size")
```

Again, we have tried to access the member of the null instance. As you can see we have used optional types of “Boolean?” and “String?”. To explicitly access members without using the safe “elvis” operator we will use the not-null assertion operator: “!!”. Using “!!” tells the Kotlin compiler that we are sure that it is safe to access instance members. Doing this may crash the application if we are wrong! The operator converts any value to a non-null type. If the value is null it throws an exception. This is why the previous snippet will crash our program. Let’s get back to the “elvis” operator. What happens if we go with the safe option and use “elvis”?

```
// 'size' constant gets Int? type:
val size = nullStringVariable?.length
println("Length is: $size")
```

As it is mentioned in the comment line, constant “size” will get the optional type of “Int?”. In this case, running the snippet will make the following output but it will not crash our program:

```
Length is: null
```

Update the snippet by assigning value to the “nullStringVariable”:

```
nullStringVariable = "Elephant"
val size = nullStringVariable?.length
println("Length is: $size")
```

The output of this snippet will be:

```
Length is: 8
```

In this example, the “size” constant got “Int?” type too.

If we operate pure “Int” type and its ”optional” equivalent what will happen? Take a look at our next example:

```
nullStringVariable = "House"
val number = 10
val size = nullStringVariable?.length

val sum = size + number
```

The last line of the code snippet will not work! The compiler will complain with the following message:

“Operator call corresponds to a dot-qualified call 'size.plus(number)' which is not allowed on a nullable receiver 'size'”

To make this work we must access operator member on optional type:

```
// This will work:
val sum = size?.plus(number)
println("Sum is: $sum")
```

The output of the executed snippet will be:

```
Sum is: 15
```

The type of “sum” constant is “Int?” (optional).

To reduce unnecessary code boilerplate Kotlin comes up with more great features such as null shorthands. Let's take a look at the following example:

```
val files = File("./some_directory").listFiles()
println(files?.size ?: "empty")
```

Print function will print “empty” if there are no files, otherwise, it’s size.

In next example, we will assign value if result is not null, otherwise we throw exception:

```
val subscribers = data["subscribers"] ?: throw IllegalStateException(
        "There is no any subscribers"
)
```

One more example for not null shorthand use:

```
val player = getPlayer()
player?.play("Uriah Heep - Easy Livin.mp3")
```

As you can see, Kotlin simplifies work with instances that can have null as its value. Comparing to Java this makes the life of a developer much easier.

Note:

Optional types have been introduced in Java since version 8.

Multiple assignment

Another useful feature of Kotlin is multiple assignment. Multiple assignment allows developers to assign multiple values in one expression. Let's have a look at examples to illustrate this. From the book's code examples open the "MultipleAssignement.kt" source code file and take a look at the first example:

```
val (first, second) = arrayOf(1, 2)

println("First: $first")
println("Second: $second")
```

The output of executed code snippet is:

```
First: 1
Second: 2
```

As you can see, "first" and "second" constants have "1" and "2" as assigned values.

Let's have a look at the second example:

```
fun parameters(): List<Int> {

    return listOf(3, 5, 7)
}

val (parameter1, parameter2, parameter3) = parameters()
println("Parameters: $parameter1, $parameter2, $parameter3")
```

Thanks to the same principle we have assigned values to constants "parameter1", "parameter2", and "parameter3".

The output of executed code snippet is:

```
Parameters: 3, 5, 7
```

Type checks and smart casts

To test whether the object is an instance of the specified type check operator "is" is used. If we are checking an immutable value, there is no need to do the casting.

Open "TypeCheck.kt" from book code examples. Instance check is demonstrated in the following example:

```
/**
 * Power if double data type, otherwise throw exception
 */
@Throws(IllegalArgumentException::class)
fun power(x: Any): Double {

    if (x is Double) {
        return x.pow(2.0)
    }

    throw IllegalArgumentException(
        "This function deals only with doubles"
    )
}
```

Example function accepts arguments of any data type. However, if the passed value is Double it will return its power, otherwise, it will throw an exception.

To cast explicitly into desired data type take a look at the second example:

```
/**
* Casts explicitly into the Double data type
*/
fun powerLogger(value: Any) {

    val converted = value as Double
    try {

        val pow = power(converted)
        println("Power of $converted is: $pow")
    } catch (e: IllegalArgumentException) {

        println("Error: ${e.message}")
    }
}
```

Value is cast to String data type using “as” cast operator. Logger function accepts any data type and performs explicit conversion to String data type. The most upper type for all Kotlin classes is ”Any”. Each Kotlin class inherits it. We will talk soon about Kotlin classes and inheritance.

Let’s play a little bit with these two functions:

```
val a = 2.0
val b = 3
val c = "not a double"

val aa = power(a)
println("a power is: $aa")

try {

    val bb = power(b)
    println("b power is: $bb")
} catch (e: IllegalArgumentException) {

    println("Error: ${e.message}")
}

try {

    val cc = power(c)
    println("c power is: $cc")
} catch (e: IllegalArgumentException) {

    println("Error: ${e.message}")
}
```

The output of this will be:

```
a power is: 4.0
Error: This function deals only with doubles
Error: This function deals only with doubles
```

As you can see only “a” constant will satisfy type check condition. “b” and “c” will not as they are not of Double data type.

Let’s try out “powerLogger” function:

```
powerLogger(a)
powerLogger(b)
powerLogger(c)
```

As most data types can’t be directly cast into the other data types this will crash our program:

```
Power of 2.0 is: 4.0
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot
be cast to java.lang.Double ...
```

If the cast is not possible, the cast operator throws an exception. Because of this, this approach is not safe. If we fix our code a little bit we can execute our program without a crash:

```
/**
* Casting into the Double data type
*/
@Throws(IllegalArgumentException::class)
fun powerLogger(value: Any) {

    val converted = when (value) {
        is Int -> {

            value.toDouble()
        }
        is Double -> {

            value as Double // 'as Double' can be omitted,
                            // so no cast needed
        }
        else -> {

            throw IllegalArgumentException(
                "Unsupported data passed: $” +
                "{value::class.simpleName}"
            )
    }
}

    val pow = power(converted)
    println("Power of $converted is: $pow")
}
```

Running a new modified version will give the following output:

```
try {

    powerLogger(a)
    powerLogger(b)
    powerLogger(c)
} catch (e: IllegalArgumentException) {

    println("Error: ${e.message}")
}

Power of 2.0 is: 4.0
Power of 3.0 is: 9.0
Error: Unsupported data passed: String
```

The key is to use the data type conversion function. In our case, we have executed the “toDouble” function that will convert the Integer instance into a Double data type. All basic data types have conversion functions for other basic data types. You probably remember that we have already presented this in the “Numbers in Kotlin” section.

Let’s have a look at a couple more examples:

```
if (a is String) // If a is instance of String

if (a !is String) // If a is not instance of String

fun printIfString(a: Any) {

    if (a is String) {
        println(a) // a is automatically cast to String
    }
}
```

In the last example, we don't have to use “as” cast operator. This works for both when-expressions and while-loops (we will explain both in upcoming sections of the book). However, smart casts do not work when the compiler can't guarantee that the variable cannot change between the check and the usage.

Operators and expressions

In upcoming sections, we will be covering Kotlin operators and expressions. If you are not familiar with the meaning of operators let’s explain it in short terms. Operators represent constructs defined within programming languages that have functions like behavior. However, operators differ from functions syntactically or semantically. The most common operators are arithmetic, assignment, logic, and performing comparisons.

Equality

Kotlin recognizes two types of equality:

- Equality by reference (“===” and “!==” operators)
- Equality by structure (“==” and “!=” operators)

Let’s take a look at ”Equality.kt” example:

```
data class TestEquality(val a: String, val b: Int) {
}

val a = TestEquality("Some string", 2)
val b = TestEquality("Some string", 2)
val c = a

println("a === b: ${ a === b }")
println("a === c: ${ a === c }")
println("a !== b: ${ a !== b }") // Not equal by reference
println("a !== c: ${ a !== c }") // Not equal by reference

println("a == b: ${ a == b }")
println("a == c: ${ a == c }")
```

For the data type that we are using, we have defined a data class called “TestEquality”. We will talk more about data classes later.

Observe the output of program execution:

```
a === b: false
a === c: true
a !== b: true
a !== c: false
a == b: true
a == c: true
```

If our class is not a data class comparing using == (equality by structure) would return us false instead of true.

Arithmetic operators

Kotlin supports the following arithmetic operators:

- "+": Addition (used for string concatenation too)
- "-": Subtraction
- "*": Multiplication
- "/": Division
- "%": Modulus

To see how we can use these operators open "Arithmetic.kt" from book's code examples:

```
val a = 1
val b = 2
val c = 3
val hello = "Hello"
val world = "World"

// "+" - Addition:
val sum = a + b + c
println("a + b + c = $sum")

val concatenation = hello + " " + world
// or converted into the template:
// concatenation = "$hello $world"
println(concatenation)

// "-" - Subtraction:
```

```
val sub = c - b
println("c - b = $sub")

// "*" - Multiplication:
val multi = b * c
println("b * c = $multi")

// "/" - Division:
val div = 12 / c
println("12 / c = $div")

// "%" - Modulus:
val mod = c % b
println("c % b = $mod")
```

Running example program will output the following:

```
a + b + c = 6
Hello World
c - b = 1
b * c = 6
12 / c = 4
c % b = 1
```

As you can see, using basic arithmetic operators is very simple and intuitive.

If you are asking yourselves how operators are working we will explain that now. Let’s say that you are using the addition arithmetic operator to add two numbers “a” and “b”.

Expression ”a + b” calls ”plus” member function (“a.plus(b)”). The addition operator is overloaded to work with String values and other basic data types. Char and Boolean data types are excluded from this. If you want to support an addition operator for your custom data types, all that you have to do is overload the “plus” function. Let’s see this in practice. Open “ArithmeticOverloading.kt”:

```
class MyCustomType(val param: Int) {

    operator fun plus(what: MyCustomType): MyCustomType {
        return MyCustomType(param + what.param)
    }
}
```

We have define our custom data type that defines how addition is performed. Let’s see this in action:

```
val a = MyCustomType(5)
val b = MyCustomType(10)
val c = a + b

println("a + b = ${c.param}")
```

The output of the program will be:

```
a + b = 15
```

This is a very simple implementation that gives us very powerful possibilities.

Assignment operators

Assignment operators are used for assigning value to a variable or constant. As you probably remember we have shown how to assign values to variables and constants in previous sections of the book. So let’s refresh our memory with a few short examples:

```
val a = 1
val b = 2

var c = 10
var d = a
```

And so on.

Besides simple assignment using the “=” operator, there are a few more assignment operators that we can use. Let’s have a look at “Assignment.kt” from book code examples:

```
var a = 3
var b = 5
println("a: $a, b: $b")
```

```
a += b // Is equivalent to: a = a + b,
       // translates to: a.plusAssign(b)

println("a += b: $a")

b -= a // Is equivalent to: a = a - b,
       // translates to: a.minusAssign(b)

println("b -= a: $b")

a *= b // Is equivalent to: a = a * b,
       // translates to: a.timesAssign(b)

println("a *= b: $a")

a /= b // Is equivalent to: a = a / b,
       // translates to: a.divAssign(b)

println("a /= b: $a")

a %= b // Is equivalent to: a = a % b,
       // translates to: a.modAssign(b)

println("a %= b: $a")
```

Running this example will give us the following output:

```
a: 3, b: 5
a += b: 8
b -= a: -3
a *= b: -24
a /= b: 8
a %= b: 2
```

Unary operators

Let’s demonstrate unary operators with a simple example. Open “Unary.kt” source code file from book code examples:

```
val a = 1
val b = true
val minusA = -1
val notB = !b

println("a: $a, minus a: $minusA")
println("b: $b, not b: $notB")
```

In this example, we have defined Integer and Boolean constants. By using unary operators “-” on Integer and “!” (negation) on Boolean we have inverted its values. If we run this program the following output will be printed out:

```
a: 1, minus a: -1
b: true, not b: false
```

Most common unary operators are:
- “+”: Unary plus, with expression: “+a” that is translated to: “a.unaryPlus()”
- “-”: Unary minus, with expression: “-a” (inverts sign) that is translated to: “a.unaryMinus()”
- “!”: Not, with expression: “!a” (inverts value) that is translated to: “a.not()”.

Increment and decrement operators

To increase or to decrease value by one Kotlin gives two operators:

- “++”, Increment with expression: “++a” (or “++a”) that is translated to: “a.inc()”
- “- -”, Decrement wit expression: “- -a” (or “- -a”) that is translated to: “a.dec()”

Open “IncrementDecrement.kt” source code file from book code examples:

```
var a = 10

println("a: $a")
println("a: ${a++}")
println("a: ${++a}")
println("a: ${a--}")
println("a: ${--a}")
```

We have defined variable “a” with the value of 10. Then, we will increment it twice and decrement it twice too. Running this program will give the following output:

```
a: 10
a: 10
```

```
a: 12
a: 12
a: 10
```

If the output is not what you have expected it to be, this is because “a++” and “+ +a” do not have the same behavior. “++a” increases the value of a variable first and then returns it. While, on the other hand, “a++” returns the value of the variable first, and then increases it. The same applies to the decrease operator.

Comparisons

To compare two instances of certain data types we use comparison operators. Compared objects (instances) can be equal, object “a” can be smaller than object “b”, and opposite. Objects can be not equal as well. We will illustrate each of the commonly used comparison operators with proper examples. Open “Comparison.kt” from book code examples:

```
val a = 1
val b = 2
val c = 1

println("a: $a, b: $b, c: $c")
println("a == b: ${a == b}")
println("a == c: ${a == c}")
println("a != b: ${a != b}")
println("a != c: ${a != c}")

println("a > b: ${a > b}")
println("a > c: ${a > c}")
println("a < b: ${a < b}")
println("a < c: ${a < c}")

println("a >= b: ${a >= b}")
println("a >= c: ${a >= c}")
println("a <= b: ${a <= b}")
println("a <= c: ${a <= c}")
```

Executing our program will give us the following results:

```
a: 1, b: 2, c: 1
a == b: false
a == c: true
```

```
a != b: true
a != c: false
a > b: false
a > c: false
a < b: true
a < c: false
a >= b: false
a >= c: true
a <= b: true
a <= c: true
```

In this example we have defined three constants which are compared each other. Let's take a look at all of these comparisons that we have performed, meaning of each and corresponding functions (functions):

- ">": Greater than, with expression "a" > "b" that is translated to: a.compareTo(b) > 0
- "<": Less than, with expression "a" < "b" that is translated to: a.compareTo(b) < 0
- ">=": Greater than or equal to, with expression "a" >= "b" that is translated to: a.compareTo(b) >= 0
- "<=": Less than or equal to, with expression "a" <= "b" that is translated to: a.compareTo(b) <= 0
- "==": Equal to, with expression "a" == "b" that is translated to: a?.equals(b) ?: (b === null)
- "!=": Not equal to, with expression "a" != "b" that is translated to: !(a?.equals(b) ?: (b === null))

Logical operators

Kotlin supports two main logic operators:

- "||" known as "or" operator and
- "&&" known as "and" operator.

"Or" operator returns true Boolean if either of the Boolean expression is true, while on the other hand "And" operator returns true if all Boolean expressions are true.

Let's have a look at examples that will illustrate this for us. Open "Logical.kt" from book code examples:

```
val a = 1
val b = 2
val c = 3
val d = 4

// 'Or' examples:
val or1 = (a > b) || (c > d)
val or2 = (a < b) || (c < d)
val or3 = (a == b) || (c == d)
val or4 = (a == b) || (c == d) || (a == c)

println("($a > $b) || ($c > $d) -> $or1")
println("($a < $b) || ($c < $d) -> $or2")
println("($a == $b) || ($c == $d) -> $or3")
println("($a == $b) || ($c == $d) || ($a == $c) -> $or4")

// 'And' examples:
val and1 = (a > b) && (c > d)
val and2 = (a < b) && (c < d)
val and3 = (a == b) && (c == d)
val and4 = (a == b) && (c == d) && (a == c)

println("($a > $b) && ($c > $d) -> $and1")
println("($a < $b) && ($c < $d) -> $and2")
println("($a == $b) && ($c == $d) -> $and3")
println("($a == $b) && ($c == $d) && ($a == $c) -> $and4")

// Mixed example
val mixed = ((a < b) || (c < d)) && (a != c)
println("(($a < $b) || ($c < $d)) && ($a != $c) -> $mixed")
```

Executing the program will give us the following output:

```
(1 > 2) || (3 > 4) -> false
(1 < 2) || (3 < 4) -> true
(1 == 2) || (3 == 4) -> false
(1 == 2) || (3 == 4) || (1 == 3) -> false
(1 > 2) && (3 > 4) -> false
(1 < 2) && (3 < 4) -> true
(1 == 2) && (3 == 4) -> false
(1 == 2) && (3 == 4) && (1 == 3) -> false
((1 < 2) || (3 < 4)) && (1 != 3) -> true
```

Please go through the example’s source code line by line and compare expressions with the program’s output result. You will see that the “Or” and “And” operators behave exactly as we described them. Also, you can see that we can combine them to get more complex expressions like in the last “mixed” example.

Operator overloading

In Kotlin it is possible to create our implementations for operators. For example for “+” operator, for “-” operator, and so on. To make this happen, it is required to provide a member function or an extension function with a fixed name, for the corresponding type. Each function that overloads the operator must be marked with the “operator” modifier.

Let’s show you a simple example of “+” operator overloading. From book code examples open ”OperatorOverloading.kt”:

```
class Example(var value: String) {

    operator fun plus(what: Example): Example {

        return Example("$value, ${what.value}")
    }

    operator fun plus(what: String): Example {

        return Example("$value, $what")
    }

    override fun toString() = value
}
```

We have defined the implementation for “+” operator that can work with “Examples” class type itself and strings. Let’s try it out:

```
val a = Example("A")
val b = Example("B")
val c = Example("C")

val ab = a + b
val abc = a + b + c
val a2 = a + "Hello"
val a3 = a + "Hello" + "World"

val bc = b + "Hello" + c
val bc2 = b + c + "Hello"

println("a + b = $ab")
println("a + b + c = $abc")
println("a + 'Hello' = $a2")
println("a + 'Hello' + 'World' = $a3")
println("b + 'Hello' + c = $bc")
println("b + c + 'Hello' = $bc2")
```

If we execute this the following result will be generated:

```
a + b = A, B
a + b + c = A, B, C
a + 'Hello' = A, Hello
a + 'Hello' + 'World' = A, Hello, World
b + 'Hello' + c = B, Hello, C
b + c + 'Hello' = B, C, Hello
```

The following list contains pairs of expression name to operator function name:

+a → a.unaryPlus()
-a → a.unaryMinus()
!a → a.not()
a++ → a.inc()
a-- → a.dec()
a + b → a.plus(b)
a – b → a.minus(b)
a * b → a.times(b)
a / b → a.div(b)
a % b → a.mod(b)
a in b → b.contains(a)
a !in b → !b.contains(a)
a[i] → a.get(i)
a[i, j] → a.get(i, j)
a[i_1, ..., i_n] → a.get(i_1, ..., i_n)
a[i] = b → a.set(i, b)
a[i, j] = b → a.set(i, j, b)

a[i_1, ..., i_n] = b → a.set(a_1, ..., a_n, b)
a() → a.invoke()
a(i) → a.invoke(i)
a(i_1, ..., a_n) → a.invoke(a_1, ..., a_n)
a+=b → a.plusAssign(b)
a-=b → a.minusAssign(b)
a*=b → a.timesAssign(b)
a == b → a?.equals(b) ?: b === null
a !=b → !(a?.equals(b) ?: b ===null)
a>b → a.compareTo(b) > 0
a<b → a.compareTo(b) < 0
a>=b → a.compareTo(b) >= 0
a<=b → a.compareTo(b) <= 0

If you ever need to override some other operator rather than "+" in this you can find the name of the corresponding operator function.

Conditional expressions

Conditional expressions are features that perform different computations or actions depending on whether a programmer-specified boolean condition evaluates to true or false.

Let's examine a simple example from the "Conditional.kt" book code examples source code file:

```
fun compare(x: Int, y: Int): Int {

    if (x > y) {

        return 1
    } else if (x < y) {

        return -1
    }

    return 0
}
```

This example demonstrates the use of the "If/Else" feature. The function takes two arguments of the Integer type. If the first parameter is bigger than the second parameter function will return a value of 1. If it is smaller, it will return -1. If they are equal to each other function will return 0.

Same function can be simplified as ”one-liner”:

```
fun compare(x: Int, y: Int) =
    if (x > y) 1 else if (x < y) -1 else 0
```

Let’s run this function and see what output it will be produced:

```
var x = 1
var y = 2
println("Compare $x vs $y: ${compare(x, y)}")

x = 2
y = 1
println("Compare $x vs $y: ${compare(x, y)}")

x = 2
y = 2
println("Compare $x vs $y: ${compare(x, y)}")
```

Execution output of the program is:

```
Compare 1 vs 2: -1
Compare 2 vs 1: 1
Compare 2 vs 2: 0
```

If expression

In Kotlin “If” is an expression and it can return a value. We will examine the use of “If” on “If.kt” book’s code source example:

```
fun getMemberTypeById(id: Int): String {

    return if (id == 0) {
        "Unregistered"
    } else if (id == 1) {
        "Registered"
    } else if (id == 2) {
        "Admin"
    } else {
        "Unknown"
    }
}
```

We have defined a function that will accept the user’s id as an argument of Integer type and return associated type with that Id. Let’s try it:

```
val ids = listOf(0, 1, 2, 3, 4)

ids.forEach {

    val memberTyeId = getMemberTypeById(it)
    println("Id: $it -> $memberTyeId")
}
```

We have defined a list with Ids from 0 to 4. We will iterate through the list and print member types for each of Ids. Executing the program will give us the following output:

```
Id: 0 -> Unregistered
Id: 1 -> Registered
Id: 2 -> Admin
Id: 3 -> Unknown
Id: 4 -> Unknown
```

Same function can be written with less code:

```
fun getMemberTypeById(id: Int) = if (id == 0) {

    "Unregistered"
} else if (id == 1) {

    "Registered"
} else if (id == 2) {

    "Admin"
} else {

    "Unknown"
}
```

or rewritten into “When” expression (Kotlin’s answer to traditional “Switch/Case”):

```
fun getMemberTypeById(id: Int) = when (id) {

    0 -> {
        "Unregistered"
    }

    1 -> {
        "Registered"
    }

    2 -> {
        "Admin"
    }

    else -> {
        "Unknown"
    }
}
```

See “When.kt” from book code examples.

When

Unlike “If” and ”If/Else”, the ”When” statement can have a number of possible execution paths. The “When” statement replaces the traditional ”Switch” operator that we used in Java. “When” can be used either as an expression or as a statement.

“When” matches its argument against all branches sequentially. Matching is performed until a branch condition is satisfied. If it is used as an expression, the value of the satisfied branch becomes the value of the overall expression. The values of individual branches are ignored if it is used as a statement.

If none of the other branch conditions are satisfied, the ”Else” branch is evaluated. If ”When” is used as an expression, the ”Else” branch is mandatory, unless the compiler can prove that all possible cases are covered with branch conditions. If many cases should be handled in the same way, the branch conditions can be combined with a comma.

“When2.kt” from book code examples illustrates common use cases of “When”:

```
fun dataTypeRecognizer(what: Any) {

    when (what) {
        is Float -> println("Floating point")
        is Int -> println("Number")
        is String -> println("String")
        is Boolean -> if (what) {
            println("Boolean, true")
        } else {
            println("Boolean, false")
        }
        else -> println("Unknown")
    }
}
```

The first function uses “When” to recognize the data type for provided argument. Now let’s have a look at the example of the “When” statement that returns a value:

```
@Throws(IllegalArgumentException::class)
fun getCarPrice(model: String) = when (model) {

    "Mercedes" -> 100
    "BMW" -> 200
    "Opel" -> 300

    else -> throw IllegalArgumentException(
        "We do not recognize this model."
    )
}
```

“When” is used to return the price for a particular car model. If none of the recognized models is found exception is thrown.

More simple examples of “When” use cases:

```
@Throws(IllegalArgumentException::class)
fun validateUserType(userType: Int) {

    when (userType) {
        0 -> println("Registered user")
        1 -> print("Administrator")
        else -> {

            println("Error recognizing user type")
            throw IllegalArgumentException(
                "Invalid user type: $userType"
            )
        }
    }
}
```

This function illustrates the use of “When” for validating user types in some imagined system. Something similar is demonstrated with our next example function (function):

```
fun validateAccountType(accountType: Int) {

    when (accountType) {
        0, 1 -> println("Welcome")
        else -> println("Permission denied")
    }
}
```

To illustrate everyday use cases we will build on top of validating user types example:

```
fun filterUserType(userType: Int): Int {

    if (userType in 0..1) {
        return userType
    }
    return -1
}

@Throws(IllegalArgumentException::class)
fun validateUserTypeFiltered(userType: Int) {

    when (userType) {
        filterUserType(userType) -> {

            println("Ok")
            validateUserType(userType)
        }
        else -> print("Not ok")
    }
}
```

And our last example function uses “When” to illustrate how precise player hits the target in a computer shooting game:

```
fun precisionCheck(points: Int) {

    val max = 100
    when (points) {
        in max / 5..max / 4 -> println("PRECISE")
        in (max / 4) until max -> println("VERY PRECISE")
        max -> println("STRAIGHT IN TARGET")
        else -> println("MISSED")
    }
}
```

Now when we have defined all these functions we will play a little bit with each of them. We will start with the first one, to check data type for passed argument:

```
println("Data type recognizer:")
listOf(
        "Hi", "there",
        1, 10, 1.10,
        5 == "5".toInt()
).forEach {

    println("$it is:")
    dataTypeRecognizer(it)
    println()
}
```

When executed this code snippet will give us the following output:

```
Data type recognizer:

'Hi' is:
```

```
String

'there' is:
String

'1' is:
Number

'10' is:
Number

'1.1' is:
Unknown

'true' is:
Boolean, true
```

Next example uses “getCarPrice” function to determine and print price for various car models:

```
println("Car prices:")
listOf("Mercedes", "BMW", "Opel", "Fiat").forEach {

    try {

        val price = getCarPrice(it)
        println("Price for $it car is: $price")
    } catch (e: IllegalArgumentException) {

        println("$it: ${e.message}")
    }
}
```

Running the code snippet will print out the following:

```
Car prices:
Price for Mercedes car is: 100
Price for BMW car is: 200
Price for Opel car is: 300
Fiat: We do not recognize this model
```

As you can see “When” matched all “supported” models and thrown exception for the first model that we do not recognize.

Next example that we will play with is with validating user types in some imagined system:

```
val user1 = "john.smith"
val user2 = "dr.cooper"
val admin = "root"
val guest = "guest"

val systemUsers = mapOf(user1 to 0, user2 to 0, admin to 1)
val challenges = listOf(user1, user2, guest, admin)

println()
println("Checking users:")
challenges.forEach {

    var userType = -1
    systemUsers[it]?.let { type ->
        userType = type
    }
    println("Checking user: $it")
    try {

        validateUserType(userType)
    } catch (e: IllegalArgumentException) {

        println(e.message)
    }
    println()
}
```

If we execute this code snippet, we will see that users “john.smith”, “dr.cooper” and “root” are recognized as valid in the system, where “guest” user is not:

```
Checking users:
Checking user: john.smith
Registered user

Checking user: dr.cooper
Registered user
```

```
Checking user: guest
Error recognizing user type
Invalid user type: -1

Checking user: root
Administrator
```

And last, but not least, our “precisionCheck” example:

```
println("Precision check:")

listOf(10, 20, 30, 40, 50, 80, 100).forEach { points ->

   println("Points: $points, precision: ")
   precisionCheck(points)
   println()
}
```

If we run this snippet and validate our points the following result will be printed out:

```
Precision check:
Points: 10, precision:
MISSED

Points: 20, precision:
PRECISE

Points: 30, precision:
VERY PRECISE

Points: 40, precision:
VERY PRECISE

Points: 50, precision:
VERY PRECISE

Points: 80, precision:
VERY PRECISE

Points: 100, precision:
```

We have validated points range from 10 to 100 and depending on “When” conditions matching proper precision status are printed out.

Classes

“Class” represents a "blueprint" for creating objects. Each object is created by calling the proper class constructor function which returns an object. In Kotlin class is defined using the “class” word.

Let’s go through “Classes.kt” from book code examples and see how we can define a class and instantiate it.

Simple class definition:

```
class Dummy {}
```

Main elements that consist the class are the following:

- the class name,
- the class header (containing its type parameters, the primary constructor, and so on),
- the class body, surrounded by curly braces.

Both the header and the body are optional. If there is no class body like in the previous example, we can omit braces:

```
class NoBody
```

Having instance of these classes can be performed like this:

```
val d1 = Dummy()
val d2 = Dummy()
val d3 = Dummy()

val nbd1 = NoBody()
val nbd2 = NoBody()
val nbd3 = NoBody()
```

In this example, we have created three unique instances of “Dummy” and “NoBody” classes. For each instantiation, we have called fault constructor which does not accept any arguments. We will talk about class constructors in the next section, so let’s see what constructors are and how to use them.

Constructors

Constructor represents a special function that is used to initialize objects. The constructor is called when an object of a class is instantiated. In Kotlin, the class can have a primary constructor and one or more secondary constructors. The primary constructor is a part of the class header. **The constructor keyword can be omitted If the primary constructor does not have any annotations or visibility modifiers.**

Let’s take a look at a couple of examples. We will continue with the “Classes.kt” source code file:

```
// Class 'Car' with primary constructor
// that accepts one argument:
class Car constructor(val brand: String)

// 'constructor' keyword is omitted for primary constructor:
class Plane(val brand: String)
```

Let’s create a few instances of these classes:

```
val car = Car("BMW")
val car2 = Car("Mercedes")
val plane = Plane("Boeing")
```

The primary constructor can't contain any code. Initialization code can be placed in initializer blocks. They are prefixed with the “init” keyword.

The next example illustrates this:

```
class Calculator(val parameter1: Int) {

   val parameter2: Int
   init {

      parameter2 = parameter1 * 2
   }
}
```

This code can be simplified:

```
class Calculator(
    val parameter1: Int,
    val parameter2: Int = parameter1 * 2
)
```

Creating a couple of instances of “Calculator” class:

```
val calc = Calculator(2)
val calc2 = Calculator(3)
val calc3 = Calculator(2, 4)
val calc4 = Calculator(3, 5)
```

If we print out every instance’s property values:

```
println("${calc.parameter1}, ${calc.parameter2}")
println("${calc2.parameter1}, ${calc2.parameter2}")
println("${calc3.parameter1}, ${calc3.parameter2}")
println("${calc4.parameter1}, ${calc4.parameter2}")
```

The following output will be produced:

```
2, 4
3, 6
2, 4
3, 5
```

Let’s say that we don't want the class to be constructed publicly, we can do it the following way:

```
class NotPublicConstructed
private constructor(val name: String) {

    // ...
}
```

Class constructor is now private. That means that it is only accessible within the class scope and not outside. Private constructors are usually used in singleton pattern implementations.

Note:

Visibility modifiers are used to restrict the accessibility of classes, objects, interfaces, constructors, functions, properties, and their setters to a certain level. There is no need to set the visibility of getters. They have the same visibility as the property. Kotlin distinguished four visibility modifiers:

- public: visible everywhere
- protected: visible inside the same class and its subclasses
- private: visible inside the same class only
- internal: visible inside the same module.

Secondary constructors

In Kotlin you can use more constructors with a different set of parameters along with the primary constructor. We will continue where we left last time and illustrate the use of secondary constructors. From book code examples open “Classes.kt”:

```
class Person(val name: String) {

    constructor(name: String, year: Int)
        : this(name) // ← calls primary constructor

    constructor(name: String, year: Int, height: Int)
        : this(name) // ← calls primary constructor
}

val person1 = Person("John Smith")
val person2 = Person("John Doe", 1985)
val person3 = Person("John SomeOther", 1987, 190)
```

The code demonstrates the usage of secondary constructors in Kotlin. If we didn’t define a primary constructor, the class will have a default–empty constructor, with no arguments.

We instantiated three instances of the “Person” class. Each with a different constructor is used for instantiation. You can also notice that Kotlin does not have the word “new” for instantiation, unlike Java.

Note:

It is required to call the primary constructor from the secondary constructor explicitly. It is important to note too that the property of the class can't be declared inside the secondary constructor.

Class members

In Kotlin classes can have the following members:

- Constructors
- Initialization blocks
- Functions
- Properties
- Nested classes
- Inner classes
- Misc object declarations – for example: "companion object".

We will talk about all of these in the upcoming sections of this book.

Class properties

One of the most important ideas behind work with classes is the concept of "encapsulation". This idea is used to encapsulate code and data into a single entity. A property in a class is declared the same as declaring a variable with "val" and "var" keywords. A property declared as var is mutable and thus, can be changed.

Let's have a look at a simple example that illustrates this. From book code examples open "Properties.kt":

```
class Coordinate {

  var x: Int = 0
  var y: Int = 0
  var z: Int = 0

  fun print() = println("Coordinate(x=$x, y=$y, z=$z)")
}

val center = Coordinate()
center.print()
```

```
val location = Coordinate()
location.x = 10
location.y = 20
location.z = 30
location.print()
```

Properties defined as “var” are mutable. Read-only properties are defined as “val”.

Let’s go one step further. We will define the means for getting and setting our encapsulated properties. From book code examples open “GetterAndSetter.kt”:

```
class Coordinate {

   var reposition: Int = 0

   var x: Int = 0
      get() = field
      set(value) {
         field = reposition(value)
      }

   var y: Int = 0
      get() = field
      set(value) {
         field = reposition(value)
      }

   var z: Int = 0
      get() = field
      set(value) {
         field = reposition(value)
      }

   fun print() = println("Coordinate(x=$x, y=$y, z=$z)")

   private fun reposition(param: Int) = param + reposition
}
```

“get” and “set” functions (functions) define how we will write and read the value of our properties. In this example “set” function performs a certain “correction” to the passed value, while on the other hand “get” function just returns the current value of the property. Because of this, in this case, the “get” function is redundant. We may get rid of it:

```
class Coordinate {

    var reposition: Int = 0

    var x: Int = 0
        set(value) {
            field = reposition(value)
        }

    var y: Int = 0
        set(value) {
            field = reposition(value)
        }

    var z: Int = 0
        set(value) {
            field = reposition(value)
        }

    fun print() = println("Coordinate(x=$x, y=$y, z=$z)")

    private fun reposition(param: Int) = param + reposition
}
```

Let’s try out our new class implementation:

```
val center = Coordinate()
center.print()

val location = Coordinate()
location.reposition = 100
location.x = 10
location.y = 20
location.z = 30
location.print()
```

Executing the program will produce the following output:

```
Coordinate(x=0, y=0, z=0)
Coordinate(x=110, y=120, z=130)
```

Inheritance

In this section, we will explain and demonstrate what inheritance is. Simply put inheritance is the process where one class acquires the properties (functions and fields) of another.

The class which inherits the properties of others is known as “subclass” (“derived class”, “child class” and so on) and the class whose properties are inherited is known as “superclass” (“base class”, ”parent class”).

All classes at the top of the hierarchy are inheriting “Any” superclass. From ”Any” class we inherit the following functions (functions):

- “equals”
- “hashCode”
- “toString”

“equals” function

Indicates whether some other object is "equal to" this one. Implementations must fulfill the following requirements:

- Reflexive: for any non-null value x, x.equals(x) should return true.
- Symmetric: for any non-null values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- Transitive: for any non-null values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- Consistent: for any non-null values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
- Never equal to null: for any non-null value x, x.equals(null) should return false.

“hashCode” function

Returns a hash code value for the object. The general contract of hashCode is:
- Whenever it is invoked on the same object more than once, the hashCode function must consistently return the same integer, provided no information used in equals comparisons on the object is modified.

- If two objects are equal according to the equals() function, then calling the hashCode function on each of the two objects must produce the same integer result.

“toString” function

Returns a string representation of the object.

We will demonstrate inheritance with simple examples. From book code examples open “Inheritance.kt”:

```
// Class 'Human' Inherits  by default: 'Any':
open class Human

// Class 'Indian' inherits Human:
class Indian : Human()

// Class with non-empty constructor:
open class Vehicle(type: String)

// Class 'Truck' inherits 'Vehicle' and it's constructor:
class Truck(type: String) : Vehicle(type)

// Class 'Train' inherits class 'Vehicle'
// but it has empty constructor.
// Value is passed to the parent constructor:
class Train : Vehicle("Civil")

// Another way to inherit class"
class Bus : Vehicle {

    // Constructor goes here:
    constructor(type: String) : super(type)

    init {

        // Your special initialization code
        // ...
    }
}
```

Note:

Class can't be inherited until it is defined as "open". By default, all Kotlin classes are closed for inheritance. Also, it is possible to inherit only one superclass. On the other hand, multiple interfaces can be implemented. We will explain this in upcoming sections of this book.

Overriding

If a subclass provides the specific implementation of the function that has been declared by its parent class, this is called function (function) overriding. Function overriding is used to provide the specific implementation of a function that is already provided by its superclass.

Note:

The function must have the same name as in the parent class and it must have the same parameter as in the parent class

In Kotlin to override function in the class we must define it as "open" in our "super" (parent) class. Let's have a look at an example that illustrates this. From book code examples open "Overriding.kt":

```
open class Engine(protected val model: String) {

    open fun turnOn() = println("$model: Turning on")

    open fun turnOff() = println("$model: Turning off")
}

class CarEngine(model: String) : Engine(model) {

    override fun turnOn() {

        super.turnOn()
        println("$model: Car is starting")
    }

    override fun turnOff() {

        super.turnOff()
        println("$model: Car is stopping")
    }
}

class CustomEngine(model: String) : Engine(model) {

    // We override just 'turnOn' function for this class:
    override fun turnOn() {

        // We don't want super class business logic,
        // so we do not call 'super':
        println("$model: Car is starting")
    }
}
```

Let’s play with these classes:

```
val carEngine = CarEngine("Fiat")
carEngine.turnOn()
carEngine.turnOff()

val customEngine = CustomEngine("Rocket")
customEngine.turnOn()
customEngine.turnOff()
```

Executing an example program will produce the following output:

```
Fiat: Turning on
Fiat: Car is starting
Fiat: Turning off
Fiat: Car is stopping
Rocket: Car is starting
Rocket: Turning off
```

Note:

A member function that is marked with ”override” is itself “open”. If you want to prohibit re-overriding, use the “final” modifier:

```
open class TruckEngine(model: String) : Engine(model) {

    final override fun turnOn() {

        // Executing something before 'super' business logic:
```

```
        println("$model: Preparing")
        super.turnOn()
        println("$model: Engine is running")
    }

    final override fun turnOff() {

        println("$model: Preparing")
        super.turnOff()
        println("$model: Engine has stopped")
    }
}

class Scania : TruckEngine("Scania"){

    // We cannot override 'turnOn' and 'turnOff' functions
    // since they are marked with 'final override'
}
```

As you can see from these simple examples, overriding is one of the most important features of object-oriented development. Overriding increases the usability of our code and gives us greater flexibility in development.

Object-oriented features

In this section, we will continue our journey through the world of object-oriented development. We will demonstrate some very important Kotlin features such as:

- Data classes
- Basics of abstraction
- Companion object
- Interfaces.

Data classes

Kotlin has a rich set of idioms to offer to developers. One of such is ”data classes”. “Data classes” are classes whose main purpose is to hold the data. In such types of classes, some standard functionalities are provided.

”Data classes” are classes marked with the “data” keyword. For “data classes” the compiler automatically derives members from all properties declared in the primary constructor. These members are:

- automatic getters and setters
- “equals” and “ hashCode” pair
- “toString” implementation that returns “String” in human-readable form
- “componentN” functions corresponding to the properties in their order of declaration
- “copy” function.

The following rules must be fulfilled to utilize “data classes”:

- The primary constructor must have one or more parameters
- All primary constructor parameters must be marked as the constants or variables
- “Data classes” cannot be “abstract”, “open”, “sealed” or “inner”

If you need to create some “DTO” (“Data Transfer Objects”) objects, then ”data classes” are ideal for you. Let’s have a look at the example the illustrates use of “data classes”. From book code examples open “Data.kt”:

```
data class User(val name: String, val age: Int)
```

Let’s instantiate a few “users”:

```
listOf(

   User("John Doe", 40),
   User("John Lord", 45),
   User("John Smith", 35)
).forEach {

   println("$it")
}
```

Executing this code snippet will produce the following output:

```
User(name=John Doe, age=40)
User(name=John Lord, age=45)
User(name=John Smith, age=35)
```

You may see now that the “toString” function produced the human-readable form of every “User” class instance.

Abstraction

“Abstract class” represents the class that cannot be instantiated and can be only inherited. Properties and functions of “abstract class” are non-abstract unless you explicitly declare them as ”abstract” by using the “abstract” keyword. Each “abstract” member doesn't have an implementation in its mother class.

Note:

We do not need to annotate an abstract class or function with an “open” modifier. It comes without saying. The same applies to “interfaces”. Also, we can override a non-abstract open member with an abstract one.

Let’s have a look at a simple abstraction example. From book code examples open “Abstraction.kt”:

```
abstract class Animal {

    abstract fun walk()
}

open class Cat : Animal() {

    override fun walk() {

        println("Cat walks")
    }
}

class Lion : Cat() {

    override fun walk() {

        println("Lion walks")
    }
}
```

Let instantiate our cats and let them walk:

```
val cat = Cat()
val lion = Lion()

cat.walk()
lion.walk()
```

Running this code will produce the following output:

```
Cat walks
Lion walks
```

As we already mentioned, in Kotlin we can make class fields abstract as well. Depending on if these fields are val or var we must implement proper getter or getter with a setter in class that is extending:

```
abstract class DbProvider {

    abstract val database: String
}

class PostgresProvider : DbProvider() {

    override val database: String
        get() = "Postgres"
}

class MySQLProvider : DbProvider() {

    override val database: String
        get() = "MySQL"
}
```

Let’s try them:

```
val databases = listOf(

    PostgresProvider().database,
    MySQLProvider().database
)

databases.forEach {
    println(it)
}
```

Printing each database value will produce the following output:

```
Postgres
MySQL
```

As you can see, overriding abstract properties is very simple.

Object

In the modern development world, one of the most frequently used design patterns is ”Singleton”. “Singleton” is an object-oriented design pattern where a certain class can have only one object (instance).

Kotlin provides an elegant solution for ”singletons”. To create a “singleton” it is required to use the “object” declaration. Object declaration is done by using the “object” keyword. To better understand this open “Object.kt” from book code examples:

```
import java.awt.Image
import java.net.URL

object ImageManager {

    private val images = mutableListOf<Image>()

    fun download(url: URL): Boolean {

        // Perform image download procedure
        // and add to 'images' collection ...
        return true
    }

    fun getImages() = images
}
```

Let’s try our ”object” (“singleton”):

```
val url = URL(
    "https://fundamental-kotlin.com/images/cover.png"
)

// No instantiation needed for the 'ImageManager' data type:
ImageManager.download(url)
```

Here we have only one instance of “ImageManager” in our program memory.

Class companion object

If you need to write a function that can be called without having a ”class” instance but needs access to the internals of a “class”, you can write it as a member of an ”object” declaration inside that “class”. If you declare a “companion object” inside your “class”, you’ll be able to call its members with the same syntax as calling “static” functions in Java, using only the “class” name as a qualifier.

Let’s take a look at “Companion.kt” from book code examples:

```
class Greeting {

    companion object {

        fun hello() = println("Hello!")
    }
}
```

Call the static function:

```
fun main() {

    // We are accessing to 'hello' function
    // without instantiation needed:
    Greeting.hello()
}
```

Console output:

```
Hello!
```

We have access to the “hello” function without making a new instance of the “Greeting” class.

Sometimes we need to create an instance of a slight modification of some (abstract) class, without explicitly declaring a new subclass for it. Kotlin handles this case with object expressions and object declarations. Let's take a look at our next example. We will create a couple of “anonymous” objects. From book code examples open “AnonymousObject.kt”:

```
abstract class Command {

    abstract fun execute()
}

class Executor : Command() {

    private val commands = mutableListOf<Command>()

    fun addCommand(command: Command) {

        commands.add(command)
    }

    override fun execute() {

        for (command in commands) {

            command.execute()
        }
    }
}
```

Now when we have defined some basic classes, we will try to define some “anonymous” objects:

```
// Let's try it:
val executor = Executor()

val obj = object : Command() { // ← Defining “anonymous”
                               // object

    override fun execute() {

        println("Starting session.")
    }
}
executor.addCommand(obj)

val obj2 = object : Command() {

    override fun execute() {

        println("Logging in.")
    }
}
executor.addCommand(obj2)

val obj3 = object : Command() {

    override fun execute() {

        println("Launch app 1")
    }
}
executor.addCommand(obj3)

val obj4 = object : Command() {

    override fun execute() {

        println("Launch app 2")
    }
}
executor.addCommand(obj4)

val obj5 = object : Command() {

    override fun execute() {

        println("Logging out")
    }
}
executor.addCommand(obj5)

val obj6 = object : Command() {
    override fun execute() {

        println("Stopping session")
    }
}
executor.addCommand(obj6)

executor.execute()
```

Executing this program will produce the following output:

```
Starting session.
Logging in.
Launch app 1
Launch app 2
Logging out
Stopping session
```

“Class companion object” can have a name. From book code examples open “NamedCompanion.kt” and have a look:

```
class Database(val type: String) {

    companion object Factory {

        fun createInMemoryDatabase() = Database("InMemory")

        fun createFilesystemDatabase() =
            Database("Filesystem")
    }

    fun describe() = println("This is '$type' database")
}
```

Let’s create some “Database” instances:

```
val inMemory = Database.Factory.createInMemoryDatabase()
val filesystem = Database.Factory.createFilesystemDatabase()
```

which can be simplified to:

```
val inMemory = Database.createInMemoryDatabase()
val filesystem = Database.createFilesystemDatabase()
```

Having “companion object” named is mostly important for interoperability with Java so access to “companion object” is performed through some friendly name. For example (“Companions.java” from book code examples):

```
// This class companion object is not named:
Greeting.Companion.hello();

// Let's try named one:
Database database = Database.Factory.createInMemoryDatabase();
database.describe();
```

Finally, let’s use our databases:

```
inMemory.describe()
filesystem.describe()
```

And see the produced output:

```
This is 'InMemory' database
This is 'Filesystem' database
```

We will take a look at another example using objects. From book code examples open “MoreObjects.kt”:

```
interface Gpu { // We will talk more about interfaces
                // in upcoming sections of the book

    fun displayImage()
}

interface SoundCard {

    fun playSound()
}

open class Computer(cpuCores: Int, memoryInGigabytes: Int) {

    open val cores: Int = cpuCores
    open val memory: Int = memoryInGigabytes
}
```

These will represent foundation for defining simple “computer” configurations:

```
val computer1 = object : Computer(4, 32), Gpu, SoundCard {

    override fun displayImage() {

        println("Displaying image")
    }

    override fun playSound() {

        println("Playing sound")
    }
}

val computer2 = object : Computer(8, 64), Gpu, SoundCard {

    override fun displayImage() {

        println("Performing hardware processing")
        println("Displaying image")
    }

    override fun playSound() {

        println("Perform sound equalization")
        println("Playing sound")
    }
}
```

We have instantiated two unique computer configurations as objects. Let’s use them now in the rest of our program:

```
listOf(computer1, computer2).forEach {

    println("Cpu cores: ${it.cores}, memory: ${it.memory}GB")
    it.displayImage()
    it.playSound()
}
```

Executing it will produce the following output:

```
Cpu cores: 4, memory: 32GB
Displaying image
Playing sound

Cpu cores: 8, memory: 64GB
Performing hardware processing
Displaying image
Perform sound equalization
Playing sound
```

And, our last but not least objects example, from book code examples ”VariablesAccess.kt” will illustrate access to variables of enclosed scope:

```
var countLeft = 0
var countRight = 0
val executor = Executor()

val left = object: Command(){

    override fun execute() {

        println("Executing: LEFT")
        countLeft++
    }
}

val right = object: Command(){

    override fun execute() {

        println("Executing: RIGHT")
        countRight++
    }
}
```

As you can see “left” and “right” objects can access to “counter” variables and increase its values. If we play a little to demonstrate this:

```
executor.addCommand(left)
executor.addCommand(left)
executor.addCommand(left)
executor.addCommand(right)
executor.addCommand(right)
executor.addCommand(left)

println("Executed left: $countLeft")
println("Executed right: $countRight")
executor.execute()
println("Executed left: $countLeft")
println("Executed right: $countRight")
```

The following result will be produced:

```
Executed left: 0
Executed right: 0
Executing: LEFT
Executing: LEFT
Executing: LEFT
Executing: RIGHT
Executing: RIGHT
Executing: LEFT
Executed left: 4
Executed right: 2
```

Interfaces

“Interfaces” are abstract types that are used to specify a behavior that classes must implement. Interfaces are declared using the ”interface” keyword, and may only contain function signature and constant declarations. “Interfaces” can contain declarations of abstract functions, as well as function implementations. What makes them different from abstract classes is that interfaces cannot store states. Interfaces can have properties. However, the properties need to be abstract or to provide accessor implementations.

Interfaces cannot be instantiated, but rather are implemented. A class that implements an interface must implement all of the non-default functions described in the interface, or be an abstract class.

We will demonstrate the usage of “interfaces” by a simple example. From book code examples open “Interfaces.kt”:

```
interface Vehicle {
    fun startEngine()

    fun stopEngine()

    fun drive()
}
```

We have defined a simple “interface” called “Vehicle”. Each vehicle can:

- start the engine
- stop the engine
- and drive.

Based on this we can define a class for “Car” as the type of vehicle:

```
open class Car : Vehicle {

    override fun startEngine() {

        println("start")
    }

    override fun stopEngine() {

        println("stop")
    }

    override fun drive() {

        println("drive")
    }
}
```

And finally, we will define a concrete car brand called ”Mercedes”:

```
class Mercedes(private val model: String) : Car() {

    override fun startEngine() {

        super.startEngine()
        println("$model: start")
    }

    override fun stopEngine() {

        super.stopEngine()
        println("$model: stop")
    }

    override fun drive() {

        super.drive()
        println("$model: drive")
    }
}
```

So, let’s play a bit with our Mercedeses:

```
val m1 = Mercedes("Mercedes-Benz AMG A 35")
val m2 = Mercedes("Mercedes-Benz AMG C 63")
val m3 = Mercedes("Mercedes-Benz AMG C 63 S")
val m4 = Mercedes("Mercedes-Benz CLA 250")

val scooter = object : Vehicle {

    override fun startEngine() {

        println("Scooter: start")
    }

    override fun stopEngine() {

        println("Scooter: stop")
    }

    override fun drive() {

        println("Scooter: drive")
    }
}

val vehicles = listOf(m1, m2, m3, m4, scooter)
```

```
vehicles.forEach {

    it.startEngine()
    it.drive()
    it.stopEngine()
    println()
}
```

As you can see we have defined four Mercedeses and a scooter, all of tehm represent “Vehicle”. If we run our program the following output will be produced:

```
start
Mercedes-Benz AMG A 35: start
drive
Mercedes-Benz AMG A 35: drive
stop
Mercedes-Benz AMG A 35: stop

start
Mercedes-Benz AMG C 63: start
drive
Mercedes-Benz AMG C 63: drive
stop
Mercedes-Benz AMG C 63: stop

start
Mercedes-Benz AMG C 63 S: start
drive
Mercedes-Benz AMG C 63 S: drive
stop
Mercedes-Benz AMG C 63 S: stop

start
Mercedes-Benz CLA 250: start
drive
Mercedes-Benz CLA 250: drive
stop
Mercedes-Benz CLA 250: stop

Scooter: start
Scooter: drive
```

```
Scooter: stop
```

Properties in Interfaces

In interfaces, properties can be declared. Property can be abstract or it can provide implementations for accessors. It is important to note that properties declared in interfaces can't have backing fields. Because of this accessors declared in interfaces can't reference them.

From book code examples open “InterfaceProperties.kt”:

```
interface Process {

    val cores: Int

    val memory: Int
        get() = 64

    fun execute()

    fun getUsedResources() = "cores=$cores, memory=$memory"
}
```

As you can see “Process” interface has two properties: “cores” and “memory”, both Integer data type. Let’s implement our interface:

```
class Download(val what: URL) : Process {

    override val cores: Int
        get() = 4

    override fun execute() {

        println(
            "Downloading: $what, using(${getUsedResources()})"
        )
    }
}

class Encrypt(val what: String) : Process {
    override val cores: Int
        get() = 8

    override val memory: Int
        get() = 128

    override fun execute() {

        println(
            "Encrypting: $what, using(${getUsedResources()})"
        )
    }
}
```

We have implemented the interface in two classes: “Download” and “Encrypt”. Overriding “cores” property is mandatory since the property is abstract. However, for the “memory” property we have a default value defined which means that we do not have to override it.

Interfaces inheritance

In Kotlin interface can inherit from other interfaces. Because of this, both provide implementations and both declare new properties and functions. Classes that implement these interfaces are required to define the missing implementations. Let’s take a look at a proper example that demonstrates this. From book code examples open “InterfaceInheritance.kt”:

```
interface Device {

    val model: String
}

interface AudioDevice : Device {

    val brand: String
    val serialNumber: Long

    override val model: String
        get() = "$brand::$serialNumber"

    fun play()
}
```

```
class MusicPlayer(
    override val brand: String,
    override val serialNumber: Long
) : AudioDevice {

    override fun play() {

        println("'$model' is playing")
    }
}
```

“Device” is our first ”interface”. It is really simple. It has only one property: “model”. ”AudioDevice” device inherits “Device” interface, it overrides the “model” property and introduces two new properties and “play” function signature. “MusicPlayer” implements all this. It overrides properties in its constructor and provides “play” function implementation. Let’s instantiate it and try out instantiated objects:

```
listOf(

    MusicPlayer("Sony", 1241),
    MusicPlayer("Panasonic", 1001001),
    MusicPlayer("Sony", 1242)
).forEach {

    it.play()
}
```

Executing this will produce the following output:

```
'Sony::1241' is playing
'Panasonic::1001001' is playing
'Sony::1242' is playing
```

Overriding conflicts

If we inherit more than one implementation of the same function we conflict will occur. From book code examples open “Conflicts.kt”:

```
interface IDummy1 {
    fun doSomething() {

        val hello = hello()
        println(hello)
    }

    fun hello(): String
}

interface IDummy2 {

    fun doSomething() {

        val hello = hello()
        println(hello.reversed())
    }

    fun hello() = "Hello world"
}
```

We have defined two interfaces: “IDummy1” and “IDummy2”. Both interfaces have functions with the same signatures. Then, we will implement them:

```
class Dummy1 : IDummy1 {

    override fun hello() = "Lorem ipsum"
}

class Dummy2 : IDummy2

class Dummy3 : IDummy1, IDummy2 {

    override fun doSomething() {

        super<IDummy1>.doSomething()
        super<IDummy2>.doSomething()
    }

    override fun hello(): String {

        return super.hello()
    }
}
```

Classes “Dummy1” and “Dummy2” do not have any conflicts. However, class “Dummy3” implements both interfaces and functions conflict occurred. The good thing is that Kotlin has a solution for this. We can decide which “super” implementation to call:

```
super<IDummy1>.doSomething()
super<IDummy2>.doSomething()
```

In this example, we will use implementation from both parents. Let’s try our classes:

```
println("Dummy 1:")
val dummy1 = Dummy1()
dummy1.doSomething()

println("Dummy 2:")
val dummy2 = Dummy2()
dummy2.doSomething()

println("Dummy 3:")
val dummy3 = Dummy3()
dummy3.doSomething()
```

Instantiating these and executing the “doSomething” function on each instance will produce the following results:

```
Dummy 1:
Lorem ipsum

Dummy 2:
dlrow olleH

Dummy 3:
Hello world
dlrow olleH
```

Functional interfaces

Interfaces with only one abstract function are called “functional interfaces”. They can have multiple non-abstract members. Let’s take a look at a simple example of such an interface.

Declaring “functional interfaces” is done by using the “fun” interface modifier. From book code examples open “FunctionalInterface.kt”:

```
fun interface Executable {

    fun execute(what: String?): Boolean
}
```

As you can see, we have defined a simple interface with only one abstract member.

You can use a lambda expression instead of creating a class that implements a functional interface manually. Based on this, you can write the following code:

```
val empty = Executable { it == null || it.isEmpty() }
```

Let’s play with it:

```
listOf("", null, "Hello", "World").forEach {

    if (empty.execute(it)) {
        println("No data")
    } else {
        println(it)
    }
}
```

Executing this code snippet will give us the following results:

```
No data
No data
Hello
World
```

Nested classes

In Kotlin classes can be nested inside the other classes. Let's take a look at the example from book code examples "Nested.kt":

```
class Nested1 {

    val a = 0

    class Nested2 {

        val a = 1

        class Nested3 {

            val a = 2
        }
    }
}
```

We have defined three classes nested into each other: "Nested1", "Nested2", and "Nested3". Each class has a constant with the same name "a", but with a different value. Let's instantiate each class and print the value of the constant:

```
val n1 = Nested1()
val n2 = Nested1.Nested2()
val n3 = Nested1.Nested2.Nested3()

println("Values: ${n1.a}, ${n2.a}, ${n3.a}")
```

As you can see, to access nested classes we must go through the parent ones:

```
val n3 = Nested1.Nested2.Nested3()
```

Now, when we execute this code snippet, the following output will be produced:

```
Values: 0, 1, 2
```

Inner classes

In situations when we need access to the members of the outer class, we should mark our class as "inner". The "inner" class has a reference to an object of an outer class. However, using "inner" classes is not recommended because of potential memory leaks. Let's take a look at the example. From book code example open “Inner.kt”:

```
class A {

    val a = 100

    inner class B {

        val b = a
    }
}
```

We have defined a class “A” with a member constant “a”. Inside the “A” class we have defined “inner” “B” class which has a member constant “b”. The “b” constant takes a value of “A” class “a” constant. To be able to work with the “inner” class, both instances must be created. “A” and “B”:

```
val v = A().B()
```

If we print the value of “B” “inner” class “b” constant:

```
println("Value of b is: ${v.b}")
```

The following output will be produced:

```
Value of b is: 100
```

Functions

In upcoming sections, we will regain our learning focus on Kotlin functions. We will direct our attention to some more advanced functions related topics, such as:

- Functions invoking
- Understanding functions types
- Using named arguments
- Single-Expression functions
- Variable argument functions
- Use of spread operator
- Local functions

- Infix functions
- Tail-Recursive functions
- Anonymous functions
- Inline functions
- Lambdas and
- Closures.

Invoke

In Kotlin, all objects with an “invoke” operator defined can be invoked as a function. Open “Invoke.kt” from book code examples:

```
class Invokable {

    operator fun invoke(): Invokable {

        println("I am invoked")
        return this
    }
}
```

We have defined the “Invokable” Kotlin class that can be invoked as a function. To invoke it, we have to create a new instance of the class and execute (invoke) it:

```
val invoker = Invokable()
invoker()
```

Invoking our class instance will give us the following output:

```
I am invoked
```

Higher-order functions

In Kotlin function can be used as a data type and passed as a parameter to other functions. A function can be returned as a result of some other function as well. Let's have a look at the example of passing a function as an argument to another function.

Open “AsType.kt” from book code examples:

```
fun calculate(

    a: Int, b: Int,
    operation: (Int, Int) -> Int
): Int {

    return operation(a, b)
}
```

A function that can accept other functions as a parameter or returns them is called the ”Higher-Order function”. Instead of usual data types as a parameter to a function, we will pass other functions, anonymous function or lambda (we will talk more about lambdas later).

Let’s try it:

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

fun diff(a: Int, b: Int) = a - b

val a = 1
val b = 2

val calculateSum = calculate(a, b, ::sum)
val calculateDiff = calculate(a, b, ::diff)

println("calculate(a, b, sum) == $calculateSum")
println("calculate(a, b, diff) == $calculateDiff")
```

We will call the “calculate” function and pass “a” and “b” with values of 1 and 2 (Integer) as arguments along with the reference to another function that will be executed and its returned value returned as a result. As you can see we will try this with “sum” and “diff” functions that will perform different math operations on “a” and “b” constants. Running our program will result in the following output:

```
calculate(a, b, ::sum) == 3
calculate(a, b, ::diff) == -1
```

“::sum” and “::diff” represents a reference to a function that we are passing as the last argument to “calculate” function. Type of both passed functions to “calculate” function is: “(Int, Int) -> Int”. That means that we are passing a function that expects two Integer arguments and returns Integer as well.

What if we want to return a function from another function instead of the usual data types? As we already mentioned, that can be done too:

```
@Throws(IllegalArgumentException::class)
fun getCalculationStrategy(strategyId: Int): (Int, Int) -> Int

{

    val strategies = getCalculationStrategies()
    if (strategyId >= strategies.size) {

        throw IllegalArgumentException(
            "No strategy available for id: $strategyId"
        )
    }
    return strategies[strategyId]
}
```

Where “getCalculationStrategies” is defined like this:

```
fun getCalculationStrategies() = listOf(::sum, ::diff)
```

Our “getCalculationStrategy” function will return us a proper calculation strategy function based on the ”strategyId” that we provide. If “strategyId” is invalid, an exception will be thrown.

Let’s try it:

```
val strategy = getCalculationStrategy(0)
val result = strategy(a, b)
println("Calculation result is: $result")
```

Executing our program will obtain proper calculation strategy, perform strategy on “a” and “b” constants and finally print out the result:

```
Calculation result is: 3
```

Named arguments

In Kotlin function parameters can be named. Let's take a look at the example ”NamedParameters.kt” from book code examples and see how we can use this in everyday work:

```
fun logNewEntry(

    date: Date = Date(), who: String, what: String = ""
) {

    val jobPerformer = if (what.isEmpty()) {

        "No job performed"
    } else {

        "Performed: $what"
    }
    println("$date :: $who :: $jobPerformer")
}
```

This is a simple function that accepts three arguments and based on them performs some logging. The first and last arguments (parameters) are optional since we have defined default values that will be assigned to arguments if we do not provide them. Now we will try this function with several variations:

```
val today = Date()
val name = "John Smith"
val cleaning = "Cleaning windows"

logNewEntry(today, name)
logNewEntry(today, name, cleaning)
logNewEntry(date = today, who = name, what = cleaning)
logNewEntry(what = cleaning, date = today, who = name)
logNewEntry(who = name)
logNewEntry(today, who = name)
```

As you can see, we can omit arguments that have default values defined in the function’s signature. Also, what is important for this example, we can provide function arguments (parameters) by name. We have bolded and underlined arguments passing by name so you don’t miss it. Thanks to this, we are not in obligation to follow the parameters (arguments) ordering since they are passed by their name. Also, we can combine passing arguments by name with the standard way of doing this. All arguments that are not passed by name must be provided exactly in the same ordering as it is in the function's signature.

If we run this small program the following output will be printed out:

```
Sun Nov 01 12:50:34 CET 2020 :: John Smith :: No job performed
Sun Nov 01 12:50:34 CET 2020 :: John Smith :: Performed: Cleaning windows
Sun Nov 01 12:50:34 CET 2020 :: John Smith :: Performed: Cleaning windows
Sun Nov 01 12:50:34 CET 2020 :: John Smith :: Performed: Cleaning windows
Sun Nov 01 12:50:34 CET 2020 :: John Smith :: No job performed
Sun Nov 01 12:50:34 CET 2020 :: John Smith :: No job performed
```

Single-Expression functions

Functions in Kotlin can be declared as single-expression. This can help reduce our codebase size and make things simpler in some situations. Let's have a look at "SingleExpression.kt" from book code examples. We will start with regular function implementation and then we will make it simpler in a couple of steps until we get the simplest single-expression function:

```
fun calculate(a: Int, b: Int): Int {

    return a + b
}

fun calculateAsSingleExpression(

    a: Int, b: Int
): Int = a + b

fun calculateAsSingleExpressionShortest(

    a: Int, b: Int
) = a + b
```

This example illustrates the same function implemented in three different (similar) ways. The first version of the function implements the calculation function in the usual traditional way. The function has a signature that defines two Integer arguments and a function body that performs the calculation and returns the result (also Integer type).

The second version of the calculation function implements the same thing as a single expression. Instead of having a full function body, we are using just an expression that performs the same calculation as in the former version of the calculation function.

The last version of the calculation function removes the return type since it is deduced from our calculation which makes our codebase even smaller.

Variable argument functions

Defining a function with a variable number of arguments is one of the features that have many modern programming languages. Kotlin is not an exception. "Vararg.kt" illustrates how we can define such functions:

```
fun sum(vararg args: Int) {

    var sum = 0
    for (x in args) {
        sum += x
    }
    println("Sum: $sum")
}
```

"Sum" function accepts the variable number of arguments and accumulates the total value that is finally printed out. Let's try it:

```
sum()
sum(1)
sum(1, 2)
sum(1, 2, 3)
sum(1, 2, 3, 4)
```

Running this source code snippet will give us the following output:

```
Sum: 0
Sum: 1
Sum: 3
Sum: 6
Sum: 10
```

Functions with variable arguments can be combined with standard arguments too:

```
fun sumAndMultiply(multiply: Int, vararg args: Int) {

    var sum = 0
    for (x in args) {
        sum += x
    }
    val result = sum * multiply
    println("Sum and multiply: $result")
}
```

This function is very similar to the one from the previous example with one difference. We have one “fixed” additional parameter. We will again calculate the sum and then multiply it with the value of the “multiply” argument:

```
val multiply = 100
sumAndMultiply(multiply)
sumAndMultiply(multiply, 1)
sumAndMultiply(multiply, 1, 2)
sumAndMultiply(multiply, 1, 2, 3)
sumAndMultiply(multiply, 1, 2, 3, 4)
```

Running this source code snippet will give us the following output:

```
Sum and multiply: 0
Sum and multiply: 100
Sum and multiply: 300
Sum and multiply: 600
Sum and multiply: 1000
```

Spread operator

Let’s extend the second example from the previous section and make it more flexible. From book code examples open “SpreadOperator.kt”. We have modified the “sum” function to return a value:

```
fun sum(vararg args: Int): Int {

    var sum = 0
    for (x in args) {
        sum += x
    }
    return sum
}
```

Instead of having repeated code in the “sumAndMultiply” function, we will pass all our variable arguments to the “sum” function. Then, we will multiply the value as we did in the previous section:

```
fun sumAndMultiply(multiply: Int, vararg args: Int) {

    val sum = sum(*args)
    val result = sum * multiply
    println("Sum: $sum, and multiply: $result")
}
```

To pass all variable arguments to other functions that are also accepting variable arguments we have used spread operator, prefix parameter with “*”.

Let’s try new version of the function:

```
val multiply = 100
sumAndMultiply(multiply)
sumAndMultiply(multiply, 1)
sumAndMultiply(multiply, 1, 2)
sumAndMultiply(multiply, 1, 2, 3)
sumAndMultiply(multiply, 1, 2, 3, 4)
```

Which will give us the following output when it is executed:

```
Sum: 0, and multiply: 0
Sum: 1, and multiply: 100
Sum: 3, and multiply: 300
Sum: 6, and multiply: 600
Sum: 10, and multiply: 1000
```

Let’s have a look at another use case of spread operator:

```
val numbers = intArrayOf(1, 2, 3, 4, 5)
sumAndMultiply(multiply, *numbers)
```

Spread operator can be used with arrays too. If we execute this source code snippet, the output of the program will be:

```
Sum: 15, and multiply: 1500
```

All array members are used as variable arguments by the “sumAndMultiply” function. As you can see from this example, the spread operator is a great way to increase your flexibility when it comes to working with functions that are accepting a variable number of arguments.

Local functions

If we need to define a function inside the scope of one existing function, we can do it. This is possible to do in Kotlin. To illustrate how to do this we will examine an example from the “LocalFunctions.kt” source code file from book code examples:

```
fun worker(vararg jobs: String) {

    fun doWork(work: String) {

        println("Job '$work' is starting")
        println("Job '$work' is executing")
        println("Job '$work' is completed")
    }

    jobs.forEach {

        doWork(it)
        println("- - - - - - - - - - - - - - - - - -")
    }
}
```

We have defined the function “work” that accepts a variable number of arguments that represent jobs. The function executes each job by invoking a local function named “doWork”. “doWork” is local, therefore can be used only inside the “worker” function scope. If we run this small program the following output will be printed out:

```
Job 'Importing user data' is starting
Job 'Importing user data' is executing
Job 'Importing user data' is completed
- - - - - - - - - - - - - - - - - -
Job 'Processing user data' is starting
Job 'Processing user data' is executing
Job 'Processing user data' is completed
- - - - - - - - - - - - - - - - - -
Job 'Exporting user data' is starting
Job 'Exporting user data' is executing
Job 'Exporting user data' is completed
- - - - - - - - - - - - - - - - - -
```

Infix functions

One of the great features of the Kotlin programming language is Infix functions. Infix functions give us the ability to call functions with the name but by omitting the dot and the parentheses for the call followed by the argument of the function. For Infix functions the following requirements must be satisfied:

- Infix functions must be member functions or extension functions (we will talk more about extension functions later in this book)
- They are limited to a single parameter
- Function parameter must not accept the variable number of arguments (must be without default value too).

Let’s take a look at a simple example from book code examples. Open “Infix.kt” source code example:

```
infix fun Double.powerPI(x: Int): Double {

    return this.pow(Math.PI)
}
```

“powerPI” function extends “Double” data type with an additional function that can be used as “Infix”. The function accepts the Integer parameter and returns the power of that parameter by Pi (3.14). Infix functions can be used like it is illustrated in the rest of this example:

```
val array = arrayOf(2.0, 4.0, 6.0, 8.0, 10.0)
array.forEach {

    val result = it powerPI 5
    println("$it powerPI: $result")
}
```

As you can see we have executed our Infix function without dot and parentheses:

```
val result = it powerPI 5
```

One of the most frequent use cases for Infix functions is their use in unit tests.

Tail-recursive functions

Kotlin comes with support for ”tail-recursion”. ”Tail-recursion” is a replacement for loops (comes from the functional programming world). To achieve the best performance some algorithms are implemented recursively. However, with “tail-recursion”, there is a risk of a stack overflow. Stack overflow is is an error that cannot be caught and which happens when the JVM stack runs out of space. Usually, this is caused when a recursive function doesn't have the correct termination condition.

Tail recursion is a technique where the compiler can imperatively rewrite a recursive function, assuming that certain rules are met: the recursive call must be the last call of the function.

Let's take a look at one simple example that illustrates this. Open ”TailRecursive.kt” from book code examples:

```
tailrec fun tailRecursiveExample(word: String) {

    if (word.length == 1) {

        println(word)
    } else {

        println(word)
        tailRecursiveExample(

            word.substring(0..(word.length – 2))
        )
    }
}
```

In each iteration “tailRecursiveExample” function will print out one letter less of the passed argument. Let’s try it:

```
tailRecursiveExample("Hello world")
```

Which will produce the following output:

```
Hello world
Hello worl
Hello wor
Hello wo
Hello w
Hello
Hello
Hell
Hel
He
H
```

Note:

When a function is marked with the ”tailrec” the compiler optimizes the recursion. As result, we have a fast and efficient loop.

Anonymous functions

An anonymous function is similar to regular functions except that its name is omitted. Function’s body can be a block or it can be an expression.

Let's have a look at book code examples for this ”Anonymous.kt”:

```
// As block:
val f1 = fun(x: Int, y: Int): Int {

   return x + y
}

// As expression:
val f2 = fun(x: Int, y: Int) = x + y
```

As you can see we have created two anonymous functions: “f1” and “f2”. If we execute them:

```
val x = f2(1, 2)
val y = f1(3, 4)

println("x: $x")
println("y: $y")
```

We will get the following output:

```
x: 3
y: 7
```

Inline Functions

As you remember we have demonstrated the use of higher-order functions in the previous section of the book. However, using higher-order functions has certain drawbacks:

- Each function is an object
- It captures a closure
- Memory allocations and virtual calls introduce runtime overhead

To resolve this shortcoming inline functions are used. So, what are inline functions in Kotlin? Inline functions are Kotlin’s feature where the compiler inlines the function body. Compiler substitutes the body directly into occurrences where the function gets called.

Let's take a look at example of inline functions from book code examples “Inline.kt”:

```
inline fun inlined(function: () -> Unit) {

    function()
}

fun main() {

    fun hello() {

        println("Hello!")
    }

    inlined(::hello)
}
```

As you can see we have marked our “inlined” function with the “inline” modifier. Once the code snippet is executed the following output will be produced:

```
Hello!
```

Beyond the hood, all compiler magic has been performed and the final result achieved.

Now, let's consider the following situation: we are using inline function, which accepts another function that should not be inlined. Open “NoInline.kt” from book code examples:

```
inline fun noInlined(f1: () -> Unit, noinline f2: () -> Unit) {

    f1()
    f2()
}

fun main() {

    fun hello() = println("Hello")

    fun world() = println("World")

    noInlined(::hello, ::world)
}
```

As you can see, by using the “noinline” modifier second argument function will not be inlined. Executing this simple example will produce the following output:

```
Hello
World
```

As we just saw we ignored inlining using the “noinline” modifier. Again, all magic is done by the Kotlin compiler.

Note:

The inline modifier affects both the function and the lambda arguments. Everything will be inlined in the call and it will gain us better performance.

Lambdas

The Lambda function is essentially an anonymous function that can be treated as a value. Lambda function can be passed as an argument to other functions or returned as a result. In fact, with the lambda function, we can do anything that we usually do with a regular object.

So, how do we define lambda function? Let's give a simple example. Open source code file from book code examples "Lambda.kt":

```
val lambda1: (Int, Int) -> Int = { x, y -> x + y }
```

Where "(Int, Int) -> Int" is data type for constant "lambda1" and "{ x, y -> x + y }" is lambda function definition.

We can rewrite this a little bit into the second version of the function that will deduce data type from lambda function:

```
val lambda2 = { x: Int, y: Int -> x + y }
```

As you can see we did not specify the data type for constant "lambda2" explicitly. Both constants can be invoked like any other function or passed to other functions as arguments:

```
fun calculator(calculation: (Int, Int) -> Int, a: Int, b: Int)
    = calculation(a, b)
```

The "calculator" function accepts three arguments: lambda function that will be used to perform calculation, and two integers that will be used by the calculation.

If we extend this example we can introduce another function that will give us a calculation mechanism and return us lambda as a result:

```
@Throws(IllegalArgumentException::class)
fun calculationProvider(type: Int) = when(type) {

    0 -> lambda1
    1 -> lambda2
    else -> throw IllegalArgumentException(

        "Unknown type: $type"
    )
}
```

```
@Throws(IllegalArgumentException::class)
fun calculator2(a: Int, b: Int, calculationType: Int): Int {

    val calculation = calculationProvider(calculationType)
    return calculation(a, b)
}
```

“calculator2” function accepts numbers that will be used in the calculation (Integers) and identifiers based on which proper calculation will be performed. Helper function “calculationProvider” is the one that will return the lambda function based on this id.

Let’s try all this:

```
val a = 1
val b = 2
val c = lambda1(a, b)
val d = lambda2(a, b)
val e = calculator(lambda1, a, b)
val f = calculator(lambda2, a, b)
val g = calculator2(a, b, 0)
val h = calculator2(a, b, 1)

println("$a + $b with lambda1: $c")
println("$a + $b with lambda2: $d")
println("calculator with $a, $b and lambda1: $e")
println("calculator with $a, $b and lambda2: $f")
println("calculator2 with $a, $b and calculation type 0: $g")
println("calculator2 with $a, $b and calculation type 1: $h")
```

If we run our little program the following output will be produced:

```
1 + 2 with lambda1: 3
1 + 2 with lambda2: 3
calculator with 1, 2 and lambda1: 3
calculator with 1, 2 and lambda2: 3
calculator2 with 1, 2 and calculation type 0: 3
calculator2 with 1, 2 and calculation type 1: 3
```

As you can see using lambda functions may significantly increase productivity by giving great flexibility in our everyday development. Take your time and play a little bit with lambdas as using them can help you a lot regularly in the future.

Closures

A lambda expression, an anonymous function, a local function, and an object expression can access its closure. The variables are declared in the outer scope.

Let's take a look at the following example that perfectly shows this. From book code examples open “Closures.kt” and observe it:

```
val values = listOf(2, 4, 6, 8, 10)

fun calculate(): Int {

   var result = 0
   values.forEach {

      result += it
   }
   return result
}
```

As you can see calculate function can access its outer scope and iterate through the “values”. The same happens within the “foreach” block. From this block, we can access to outer scope and modify the “result” variable.

Executing:

```
println("Result is: ${calculate()}")
```

Will give us the following output:

```
Result is: 30
```

Control flow

In this section, we will continue with “control flow”. “Control flow” represents the order in which individual statements, instructions or function calls of an application are executed or evaluated. We will explain the most important Kotlin features that are used to create logic for our programs.

If expression

As we have already mentioned, in Kotlin "If" is an expression. In Kotlin "If" returns value too. Let's play a little with "If" to remind ourselves about this. From the book code example open "If.kt" located under the "control_flow" package:

```
fun check(x: Int, y: Int) {

   val result = if (x >= y) {

      println("x >= y")
      true
   } else {

      println("x is not >= y")
      false
   }
   println("Result: $result")
}
```

As you can see "result" constant receives the value from the "If" expression. If we call the "check" function with the following parameters:

```
check(2, 5)
check(2, 2)
check(5, 2)
```

The following output will be produced thanx to "If" expression's logic:

```
x is not >= y
Result: false
x >= y
Result: true
x >= y
Result: true
```

Note:

If you choose to use “If” as an expression rather than a statement, your “If” expression must provide the ”Else” part.

Loops

One of the main control flow features that we will cover is loops. Loops represent a sequential set of instructions that is repeated until a certain condition is met. Once the operation is performed, some condition is checked, for example, whether a variable has a certain value.

Kotlin has the following looping mechanisms:

- For
- While
- Do / While.

Each of these has its benefits and use cases. We will present to you simple examples for each so you get a feeling of how and when to use them.

For

“For” loop is a statement for specifying iteration, which allows code to be executed repeatedly. ”For” loop iterates through everything that provides an “iterator” and has a member, or extension-function “iterator”. It must be “Iterator” whose return type has a member, or extension-function “next”, “hasNext” that returns boolean. All of these three functions need to be marked as operators.

In this section, we will show some examples of ”for” looping. From book code examples open “For.kt”:

```
fun counter(to: Int) {

    for (x in 0..to) {
        println("x: $x ")
    }
}
```

“counter” function counts from zero to the number that we sent as a parameter. In each iteration, it prints the current value. So, for the parameter of five:

```
counter(5)
```

We get the following output:

```
x: 0
x: 1
x: 2
x: 3
x: 4
x: 5
```

If we want to achieve the same without the last number (from zero to four) we can use the “until” function instead of “..”:

```
fun counter2(to: Int) {

    for (x in 0 until to) {
        println("x: $x ")
    }
}
```

Let’s try this version of the function:

```
counter2(5)
```

And get the following output:

```
x: 0
x: 1
x: 2
x: 3
x: 4
```

As you can see these were simple examples of For looping. We could use this approach to count from 0 to the last index of collection and then access each collection element as we iterate.

If we need only indexes we can do the following:

```
fun cars(cars: List<String>) {

    for (index in cars.indices) {
        println("Car '${cars[index]}' index is: $index")
    }
}
```

This example shows direct access to indexes of collection. However, in practice, we will need more than that. We usually need more flexibility. Very frequently we will need to iterate through elements of some collection and access it more directly. Take a look at the following example:

```
fun cities(cities: List<String>) {

    for (city in cities) {
        println("City: ${city.capitalize()}")
    }
}
```

“cities” function iterates through the collection using Java’s “foreach”. Let’s rewrite it Kotlin way:

```
fun citiesKt(clubs: List<String>) {

    clubs.forEach { city ->
        println("City: ${city.capitalize()}")
    }
}
```

“citiesKt” function iterates and prints each element:

```
val cities = listOf("Belgrade", "Rome", "Moscow", "New York")
citiesKt(cities)
```

Both functions “cities” and “citiesKt” will produce the same output with “cities” collection passed as argument. If we run it we will see the following output:

```
City: Belgrade
City: Rome
City: Moscow
City: New York
```

As you can see, in “citiesKt” we have used lambda expression with values for the item itself. There is also a version of the foreEach as function without lambda expression:

```
clubs.forEach(

        Consumer { city ->
            println("City: ${city.capitalize()}")
        }
)
```

In the “cars” function example we got only indexes that we could use to access items if needed. In the “citiesKt” function example we got only items with no indexes. So, what if we need both? Well, in Kotlin, it is really easy to have it:

```
fun players(players: List<String>) {

    players.forEachIndexed { index, item ->
        println("Player $index: ${item.capitalize()}")
    }
}
```

The “players” function accepts collection of a players. Then, it iterates using Kotlin ”forEachIndexed” function. We used the lambda expression with values for item and index. It is important to note that there is a version of the same function without lambda expression too.

Let’s try the function:

```
val players = listOf("John Smith", "John Doe", "Peter Pan")
players(players)
```

Running it will produce the following output:

```
Player 0: John Smith
Player 1: John Doe
Player 2: Peter Pan
```

In the next section, we will take a look at While looping which will give us even more power when it comes to repeated code execution.

While loop

As you may already know While loop is a statement that allows code to be executed repeatedly based on a given condition. While is composed of condition or expression. When the condition or expression is evaluated, and if the condition or expression is true, the code within the block is executed. This repeats until the condition or expression becomes false. Because the while loop checks the condition or expression before the block is executed, the control structure is often also known as a pre-test loop. Compare this with the do-while loop, which tests the condition / or expression after the loop has been executed.

Let's see a simple example using the While loop. Open ”While.kt” from book code examples:

```
fun counter(to: Int) {

    var x = 0
    while (x <= to) {
        println("x: $x")
        x++
    }
}
```

The “counter” function will count from zero to a provided argument. It works by increasing the value of the counter variable “x” by one until “x” becomes less or equal to provided argument (“to”). Let’s try it:

```
counter(5)
```

Running this small program will give us the following output:

```
x: 0
x: 1
x: 2
x: 3
x: 4
x: 5
```

Do / While loop

Do / While is very similar to the While loop with one small difference. Do / While first performs the operation and then checks for the condition. To better understand this take a look at “DoWhile.kt” from book code examples:

```
fun counter(to: Int) {
    var x = 0
    do {

        println("x: $x")
        x++
    } while (x <= to)
}
```

The “counter” function prints the “x” value first and increases the counter, and then it checks if the condition is fulfilled. If it is, looping is stopped. This approach guarantees at least one execution of the main block. Let’s try our function:

```
val numbers = intArrayOf(0, 1, 2)
numbers.forEach {

    println("Counting to: $it")
    counter(it)
    println()
}
```

If we run our program, the following result will be printed out:

```
Counting to: 0
x: 0

Counting to: 1
x: 0
x: 1

Counting to: 2
x: 0
x: 1
x: 2
```

Ranges

As you already saw in our previous examples we have defined ranges to iterate through it. Let’s refine our knowledge on how to create and use ranges with more examples.

Once again we will iterate through ranges. For example, We want to iterate from zero to the number provided as an argument of the function. From book code examples open “Ranges.kt”:

```
fun range(to: Int) {

    for (x in 0..to) println("x: $x")
}
```

Let’s try it:

```
range(5)
```

Which will produce the following output:

```
x: 0
x: 1
x: 2
x: 3
x: 4
x: 5
```

Let’s revers our counting:

```
fun reverseRange(to: Int) {

    for (x in to downTo 0) println("x: $x")
}
```

Instead of iterating from zero to the number provided as an argument of the function, we will iterate from the number provided as an argument to a function towards zero:

```
reverseRange(5)
```

Running this example will produce the following output:

```
x: 5
x: 4
x: 3
x: 2
x: 1
x: 0
```

And finally, if we want to check if there is (or isn't) a member in a range we can check like in our last ranges example:

```
fun checkInRange(what: Int, to: Int) {

    if (what in 0..to) {
        println("$what is in range between 0 and $to")
    } else {
        println("$what is NOT in range between 0 and $to")
    }
}
```

Note:

"in" operator can be used to check for negation (in this case, checking if the member is not contained in range) by adding "!" (negation):

```
if (what !in 0..to) {
    ...
}
```

"checkInRange" function checks if provided function argument "what" is missing in the range between zero and "to". If we test some candidates:

```
for (x in -5..10) {

    checkInRange(x, 5)
}
```

We will get the following output:

```
-5 is NOT in range between 0 and 5
-4 is NOT in range between 0 and 5
-3 is NOT in range between 0 and 5
-2 is NOT in range between 0 and 5
-1 is NOT in range between 0 and 5
0 is in range between 0 and 5
1 is in range between 0 and 5
2 is in range between 0 and 5
3 is in range between 0 and 5
4 is in range between 0 and 5
```

```
5 is in range between 0 and 5
6 is NOT in range between 0 and 5
7 is NOT in range between 0 and 5
8 is NOT in range between 0 and 5
9 is NOT in range between 0 and 5
10 is NOT in range between 0 and 5
```

Jump expressions

Kotlin has three commonly used jump expressions:

- "return": by default it returns a value from the nearest function or anonymous function
- "break": terminates nearest enclosing loop
- "continue": goes to next step of nearest enclosing loop.

Break operator

Let's illustrate the "break" jump operator with simple code example. From book code examples open "Break.kt":

```
for (x in 0..10) {
    if (x == 5) {
        break
    }
    println("x: $x")
}
```

As you can see, once "x" gets the value of five, it will break the loop. Executing this code snippet will produce the following output:

```
x: 0
x: 1
x: 2
x: 3
x: 4
```

Continue operator

The “continue” jump operator skips the current iteration of the enclosing loop and the control of the program jumps to the end of the loop body. Continue is used to stop the execution of the body and control goes back to the next iteration of the loop.

From book code examples open “Continue.kt”:

```
for (x in 0..10) {
    if (x % 2 == 0) {
        continue
    }
    println("x: $x")
}
```

In this example, we will print out every “x” value that is odd:

```
x: 1
x: 3
x: 5
x: 7
x: 9
```

Jump operator labels

In Kotlin, any expression can be marked with a label that has the following form:

```
@label expression
```

Let’s say that we have a ”for” loop. Then, we name that “for” loop expression with a label. This makes it possible to use the label name for the ”for” loop and by doing so whenever we want to call the same “for” loop. We can just call it by the label name.

Let’s try out labels with break. From book code examples open ”JumpOperatorLabels.kt”:

```
val x = 10
val y = 3
val z = 2

myLoop@ for (a in 0..x) {
    for (b in 0..y) {

        println("$a, $b ")
        if (a == z && b == z) {

            // Does not break the current loop,
            // but the one than encloses it:
            break@myLoop
        }
    }
}
```

As you can see we will break the enclosing loop once condition is met:

```
a == z && b == z
```

Executing code snippet will produce the following output:

```
0, 0
0, 1
0, 2
0, 3
1, 0
1, 1
1, 2
1, 3
2, 0
2, 1
2, 2
```

Return with labels

In Kotlin functions can be nested, with function literals, local functions, and object expression. Thanks to qualified returns we can to return from an outer function. Probably the most important use case is returning a value from a lambda expression.

Let's take a look at "return" examples from "ReturnAtLabels.kt" from book code examples:

```
fun example1(numbers: List<Int>, breakAt: Int) {
    numbers.forEach { item ->
        if (item == breakAt) {
            return
        }
        println("Item: $item")
    }
}
```

This simple example function iterates through the array of numbers. If during the iteration it current item has the value of “breakAt” constant, iteration is stopped by returning from the parent function.

Execute the function:

```
var numbers = arrayListOf(1, 3, 5, 7, 9)
example1(numbers, 5)
```

Console output:

```
Item: 1
Item: 3
```

Next example:

```
fun example2(numbers: List<Int>) {

    numbers.forEach myNumbers@{ item ->
        if (item % 2 == 0) {
            return@myNumbers
        }
        println("Item: $item")
    }
}
```

“example2” function returns from the nearest enclosing function. Such non-local returns are supported only for lambda expressions passed to inline functions. **If we need to return from a lambda expression, we have to label it and qualify the ”return”.**

Let’s try it:

```
numbers = arrayListOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

```
example2(numbers)
```

Console output is:

```
Item: 1
Item: 3
Item: 5
Item: 7
Item: 9
```

We can achieve the same as with ”example2” implementation by using the default label for “forEach”. Take a look at our next example:

```
fun example3(numbers: List<Int>) {

    numbers.forEach { item ->
        if (item % 2 == 0) {
            return@forEach
        }
        println("Item: $item")
    }
}
```

Let’s see if it works any different than “example2” function implementation:

```
example3(numbers)
```

Console output:

```
Item: 1
Item: 3
Item: 5
Item: 7
Item: 9
```

As you can see we got the completely same result. Finally, one last possibility when it comes to “returning”, we can replace lambda expression with anonymous function and return from there:

```
fun example4(numbers: List<Int>) {

    numbers.forEach(
        fun(value: Int) {
            if (value % 2 == 0) {
                return
            }
            println("Item: $value")
        }
    )
}
```

Let’s try it:

```
numbers = arrayListOf(1, 3, 5, 7, 9)
example4(numbers)
```

Console output:

```
Item: 1
Item: 3
Item: 5
Item: 7
Item: 9
```

Collections

A collection represents a group of objects, known as its elements. Some collections allow duplicate elements and others do not. In Kotlin, some collections are mutable and some are immutable. Collections can be ordered or unordered as well.

The most significant categorization in Kotlin is between mutable and immutable collections.

Differences between mutable and immutable collections are the easiest to explain in the example of lists. Immutable lists implement interface “List<out T>” which gives the class the following functionalities: ”size” and ”get”, access to elements by position. Mutable functionalities are gained by implementing the “MutableList<T>” interface, which gives ”add”, ”addAll” and ”remove” functionalities.

In upcoming sections, we will give examples of the use of each of these collections and illustrate some basic operations that can be performed on them.

Immutable lists

The immutable list represents a generic ordered collection of elements. Functions in its interface support only read-only access to the list’s members.

Let’s take a look at the instantiation of immutable collections. In the following examples, we will be using ”Collections.kt” from book code examples as guidelines. The first example that we are going to show is the instantiation of immutable lists:

```
val numbers = listOf(2, 4, 6)
val words = listOf("Some", "Word")
```

As you can see we have defined two immutable lists. “numbers”, which contains Integer members, and “words” which contains strings. Once again, we will note that we cannot add new members to immutable lists.

Let’s have a look at the late initialization example of one immutable list:

```
lateinit var doubles: List<Double>
```

“doubles” variable will be the data type of immutable list (“List”) that contains Double data type elements. “lateinit” means that late initialization will be performed. Variable will not consume memory until it is initialized:

```
doubles = listOf(3.14, 2.16, 1.0)
```

Accessing immutable list members is performed by its position in the list:

```
val number = numbers[0]
val word = words[1]
val double = doubles[2]
val lastNumber = numbers[numbers.lastIndex]
```

Let’s see what we got:

```
println("Number: $number")
println("Word: $word")
println("Double: $double")
println("Last number: $lastNumber")
```

Printing out obtained members of the list will give us the following output:

```
Number: 2
Word: Word
Double: 1.0
Last number: 6
```

Next that we will try with immutable lists is to determine its size:

```
val doublesListSize = doubles.size
val numbersListSize = numbers.size
val wordsListSize = words.size
```

And we will print size for both immutable lists:

```
println("Doubles list size is: $doublesListSize")
println("Numbers list size is: $numbersListSize")
println("Words list size is: $wordsListSize")
```

Which will give us the following output:

```
Doubles list size is: 3
Numbers list size is: 3
Words list size is: 2
```

Besides these functionalities, there are some others very important in everyday work with lists. We can:

- check if the list contains an element
- check if the list contains all elements
- check the position of an element in the list
- check if the list is empty
- get a slice of the list
- get list indices
- get list iterator and perform iteration through all list members.

To better understand these functionalities we will illustrate them one by one.

Checking if the list contains elements:

```
for (x in 0..10) {
    if (numbers.contains(x)){
        println("Numbers collection contains: $x")
    } else {
        println("Numbers collection does not contain: $x")
    }
}
```

As you can see, we have used the “contains” function that returns Boolean true if the function argument is contained in the list. If we execute this small snippet we will get the following output:

```
Numbers collection does not contain: 0
Numbers collection does not contain: 1
Numbers collection contains: 2
Numbers collection does not contain: 3
Numbers collection contains: 4
Numbers collection does not contain: 5
Numbers collection contains: 6
Numbers collection does not contain: 7
Numbers collection does not contain: 8
Numbers collection does not contain: 9
Numbers collection does not contain: 10
```

Checking if list contains all elements:

```
val toCheck = listOf(6, 4, 2)
val toCheck2 = listOf(1, 3, 5)
val toCheck3 = listOf(4, 6, 4)

println(
    "$toCheck in $numbers: ${numbers.containsAll(toCheck)}"
)
println(
    "$toCheck2 in $numbers: ${numbers.containsAll(toCheck2)}"
)
```

“containsAll” function will check if provided argument’s members are contained in the “numbers” collection which will produce the output:

```
[6, 4, 2] in [2, 4, 6]: true
[1, 3, 5] in [2, 4, 6]: false
[4, 6, 4] in [2, 4, 6]: true
```

Checking position of the element in the list:

```
numbers.forEach {

    val position = numbers.indexOf(it)
    println("Position of $it in $numbers is: $position")
}
```

When we execute this “indexOf” function will produce for elements the following position values:

```
Position of 2 in [2, 4, 6] is: 0
Position of 4 in [2, 4, 6] is: 1
Position of 6 in [2, 4, 6] is: 2
```

Checking if the list is empty (or full):

```
val emptyList = listOf<Int>()
println("Is 'emptyList' list empty: ${emptyList.isEmpty()}")
println("Is 'numbers' list full: ${numbers.isNotEmpty()}")
```

We have two important functions for this: “isEmpty” and “isNotEmpty”. The first one returns Boolean true if the list is empty, and the other if it is not. Let’s run this snippet:

```
Is 'emptyList' list empty: true
Is 'numbers' list full: true
```

Getting slice of the list:

```
val slice = numbers.subList(0, 2)
```

“subList” function accepts two arguments: from and to index of Integer data type. The result is a view of the portion of this list between the specified from and to indexes. The returned list is backed by this list, so non-structural changes in the returned list are reflected in this list, and vice-versa. Let’s try to check the “slice”:

```
println("Slice: $slice")
```

Which will produce the following output:

```
Slice: [2, 4]
```

Getting list indexes:

```
val indices = numbers.indices
```

“indices” collection member represents the Integer range of the valid indexes for the collection. If we use it with “numbers” list for example:

```
indices.forEach {

    println("Index: $it -> ${numbers[it]}")
}
```

We will have the following output:

```
Index: 0 -> 2
Index: 1 -> 4
Index: 2 -> 6
```

As you can see we have printed out pairs index – values for each of the indexes.

Getting list iterator and perform iteration through all list members:

```
val iterator = numbers.iterator()

while (iterator.hasNext()) {

    val item = iterator.next()
    println("Number: $item")
}
```

“iterator” function returns ”Iterator” object which enables sequentially access to collection elements. We iterate through the collection as long there is a next element available. Once we have obtained the “item”, we print out the element’s value. Executing this code snippet will produce the following result:

```
Number: 2
Number: 4
Number: 6
```

Examples that we have shown represent the very basics of work with lists. The "List" class has many member functions that you can use to achieve important tasks. A complete list of all members can be found on official Kotlin language documentation for lists:

https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list/

There you will see everything that we have mentioned and a lot more. You don't have to learn all of these functions at once since you will probably learn them all as you need them and as you use them in everyday work.

Immutable maps

The map represents a collection that holds pairs of objects. The first one is the key and the second is the value for each map entry. Map data type supports efficiently retrieving the value corresponding to each key. All map keys are unique. That means the map holds only one value for each key. As we are talking about immutable maps, functions in this interface support only read-only access to the map. Reading and writing access is supported through the "MutableMap" interface.

Let's create some immutable maps. Open "Collections.kt" and locate the following code snippet:

```
val immutableMap = mapOf("something" to 1, "else" to 2)
val immutableMap2 = mapOf(Pair(1, "Plane"), Pair(2, "Car"))
```

As you can see we have used two different ways of instantiating immutable maps. "mapOf" function creates a read-only map with the specified content. Both ways that we are using to instantiate map accept a list of pairs where the first value is the key and the second is the value.

Accessing map elements is performed by key:

```
val user = "john.smith"
val id = userIds[user]
val credentials = userCredentials[user]
println("User: $user, Id: $id, Credentials: $credentials")
```

We have obtained Id and credentials for the user by providing the key with the value "john.smith" to each of these two maps. Then we have printed out these values:

```
User: john.smith, Id: 1, Credentials: 12345
```

We can check the size of the map the same way as we did with lists:

```
println("Users count in the system is: ${userIds.size}")
```

Which will produce the following output:

```
Users count in the system is: 2
```

Besides these functionalities, there are some others very important in everyday work with maps that you need. With maps we can:

- check if the map contains key or value
- get all keys or all values of the map
- check if the map is empty
- perform filtering
- perform “+” and “-” operations

To better understand these functionalities we will illustrate them one by one same as we did for lists.

Checking if map contains a key:

```
listOf("john.smith", "guest", "john.doe").forEach {

    val hasUser = userCredentials.containsKey(it)
    if (hasUser) {
        println("$it is in the system")
    } else {
        println("$it is not in the system")
    }
}
```

Checking if map contains a value:

```
val wordPairs = mapOf("Hello" to "World", "Lorem" to "Ipsum")
listOf("Elephant", "Hello", "World", "Car").forEach {

    val containsWord = wordPairs.containsValue(it)
    if (containsWord) {
        println("$wordPairs contains value $it")
    } else {
        println("$wordPairs does not contain value $it")
    }
}
```

In these examples we have used map member functions “containsKey” and “containsValue”. The first one returns true if the map contains the specified key. If the map maps one or more keys to the specified value the second one returns true.

Let’s run these code snippets. For key check we will have the following output of the program:

```
guest is not in the system
john.doe is in the system
```

Value check code snippet output:

```
{Hello=World, Lorem=Ipsum} does not contain value Elephant
{Hello=World, Lorem=Ipsum} does not contain value Hello
{Hello=World, Lorem=Ipsum} contains value World
{Hello=World, Lorem=Ipsum} does not contain value Car
```

As you can see checking maps for keys and values is simple. You do not have to iterate through the map to find out if the key or value is contained. You just need “containsKey” and “containsValue” functions for this purpose.

Checking if map is (not) empty:

```
val maps = listOf(mapOf(), wordPairs)
maps.forEach {

    if(it.isNotEmpty()) {

        println("Map: $it")
    } else {

        println("Map: empty")
    }
}
```

Function “isNotEmpty” returns Boolean true if the map contains any keys with values. Opposite to that “isEmpty” function returns Boolean true if there are no keys with values in the map.

If we execute this example the following output will be printed out:

```
Map: empty
Map: {Hello=World, Lorem=Ipsum}
```

Performing filtering:

```
userCredentials
        .filterKeys { it.contains("smith") }
        .forEach {

            println("Smith: ${it.key}")
        }
```

We will start by filtering map keys. In this example we will filter our user credentials map by keys containing the last name “smith” as the part of the key. The “filterKeys” function returns a map containing all key-value pairs with the values based on matching of the given predicate. In our case, if the key contains “smith”. Let’s run this simple code snippet:

```
Smith: john.smith
```

As you can see only “john.smith” matches our criteria.

To filter values we can use “filterValues” function:

```
userCredentials
        .filterValues { it.toInt() > 20 * 1000 }
        .forEach {

            println("User ${it.key}, Id: ${it.value}")
        }
```

This works on exactly the same principle. This time we have filtered all users whose Id is bigger than “20 000”. Running this will give us the following output:

```
User john.doe, Id: 24680
```

This time “john.doe” is the user that satisfied our filtering criteria.

```
userCredentials.filter {

    it.key.contains("smith")
            || it.value.toInt() > 20 * 100
}.forEach { (key, value) ->

    println("User matched: $key, Id: $value")
}
```

“filter” function operates on the ”Map.Entry” data type which holds key and value both. Thanks to this we are able to filter all users that contain “smith” or all users whose Id is bigger than “20 000”. Running this source code snippet will give us the following output:

```
User matched: john.smith, Id: 12345
User matched: john.doe, Id: 24680
```

We can do the same with negation using “filterNot” function:

```
userCredentials
        .filterNot { it.key.contains("doe") }
        .forEach {

            println("User '${it.key}' does not contain 'doe'")
        }
```

This will filter all users whose map key does not contain the word “doe”:

```
User 'john.smith' does not contain 'doe'
```

Performing “+” and “-” operations:

```
val mix = userCredentials +
    mapOf(
        "somebody.else" to "11111",
        "lorem.ipsum" to "22222"
)

println("'+': $mix")
```

```
val diff = mix – userCredentials.keys
println("'-': $diff")
```

We have performed addition to the “userCredentials” map by adding two more entries. Then, from the resulting map, we have subtracted “userCredentials” which gave us the map that we have added previously. We are printing out each of the resulting maps. Let’s try this example and see what output it produces:

```
'+': {
   john.smith=12345, john.doe=24680,
   somebody.else=11111, lorem.ipsum=22222
}

'-': {somebody.else=11111, lorem.ipsum=22222}
```

Immutable sets

The set represents a collection that holds a generic unordered collection of elements that does not support duplicate elements. Functions in the “Set” interface support only read-only access to the set. For reading and write access support, ”MutableSet” interface is used.

Let’s create some immutable sets. Open “Collections.kt” and locate the following code snippet:

```
val numbersSet = setOf(2, 2, 3) // It only has members
                                             // 2 and 3, no duplicates
numbersSet.forEach(::println)   // Will output: 2 and 3
```

This example creates a set of numbers containing numbers two and three with no duplicates. Then, we are printing all set members to the standard output. Pay attention to the “::” operator. As you probably remember operator “::” represents direct access to function reference.

Accessing set elements is performed:

- by element position
- by condition
- random.

Accessing set elements by position:

```
val first = numbersSet.elementAt(0)
println("First set element is: $first")
```

Which will give us the following output:

```
First set element is: 2
```

Accessing to the first and the last element of the set:

```
val first = numbersSet.first()
val last = numbersSet.last()
```

Accessing set elements by condition:

```
val cars = setOf("Fiat", "Bmw", "Audi", "Porsche", "Renault")
val findFirst = cars.first { it.length == 3 }
val findLast = cars.last { it.startsWith("Por") }
```

We have defined a new set of strings that represent car brand names. Then, we are accessing the first element that has a length of three characters and to the last element which name starts with “Por”. Let’s print out our results:

```
println("First car that has three letters is: $findFirst")
println("Last car which name starts with 'Por': $findLast")
```

Which will produce the following output:

```
First car that has three letters is: Bmw
Last car which name starts with 'Por': Porsche
```

As you can see when it comes to work with collections, sets in this particular case, Kotlin is extremely powerful! Both functions throw exceptions if no elements match the predicate. To avoid them, use “firstOrNull” and “lastOrNull” functions instead: they return null if no matching elements are found:

```
var findFirstOrNull = cars.firstOrNull { it.length == 3 }
var findLastOrNull = cars.lastOrNull { it.startsWith("Por") }

println(
    "First car that has three letters is: $findFirstOrNull"
)
```

```
println(
    "Last car which name starts with 'Por': $findLastOrNull"
)

findFirstOrNull = cars.firstOrNull { it.length == 10 }
findLastOrNull = cars.lastOrNull { it.startsWith("Cry") }

println(
    "First car that has ten letters is: $findFirstOrNull"
)
println(
    "Last car which name starts with 'Cry': $findLastOrNull"
)
```

Let’s execute this code snippet:

```
First car that has three letters is: Bmw
Last car which name starts with 'Por': Porsche
First car that has ten letters is: null
Last car which name starts with 'Cry': null
```

As you can see our set does not car brands with ten characters in the name. Also, there is no car brand name that starts with: “Cry”.

You can use the aliases for this functions if that makes more sense to you:

```
findFirstOrNull = cars.find { it.length == 4 }
findLastOrNull = cars.findLast { it.startsWith("A") }

println(
    "First car that has four letters is: $findFirstOrNull"
)
println(
    "Last car which name starts with 'A': $findLastOrNull"
)
```

If we run this, the following output will be produced as a result:

```
First car that has four letters is: Fiat
Last car which name starts with 'A': Audi
```

Accessing set elements randomly:

```
for (x in 0..5) {

    val random = cars.random()
    println("Random chosen car model: $random")
}
```

In this example, we will choose a car brand randomly five times and print out its name. Obviously, the “random” function retrieves one of the set members randomly. If you execute this code you will have output similar (not the same) like this:

```
Random chosen car model: Audi
Random chosen car model: Bmw
Random chosen car model: Audi
Random chosen car model: Fiat
Random chosen car model: Audi
Random chosen car model: Porsche
```

If we execute this program several times in the row, with each execution we will have different output:

```
Random chosen car model: Renault
Random chosen car model: Renault
Random chosen car model: Renault
Random chosen car model: Audi
Random chosen car model: Audi
Random chosen car model: Renault
```

You may notice that the “random” function may return the same item in consecutive execution.

Besides these functionalities, there are some others very important in everyday work with sets that you need. With sets we can:

- check if the set contains the element
- check if the set contains all elements
- check if the set is empty
- perform filtering
- perform “+” and “-” operations.

To better understand these functionalities we will illustrate them one by one.

Checking if set contains element:

```
val hasRenault = cars.contains("Renault")
val hasVolvo = cars.contains("Volvo")
```

The “contains” function will return Boolean if there is a member in the set that we are checking by the passed function parameter. Let’s check our results:

```
println("Has Renault: $hasRenault")
println("Has Volvo: $hasVolvo")
```

Executing this source code snippet will produce the following output:

```
Has Renault: true
Has Volvo: false
```

Checking if set contains all elements:

```
val checking = listOf(

   listOf("Renault", "Bmw"),
   listOf("Mercedes", "Bmw")
)
checking.forEach { check ->

   val result = cars.containsAll(check)
   if (result) {

      println("$check is in $cars")
   } else {
      println("$check is NOT in $cars")
   }
}
```

In this example, we have created two lists. For each of them we are checking if all members of the collection are contained in our set. Depending on the result proper output is written:

```
[Renault, Bmw] is in [Fiat, Bmw, Audi, Porsche, Renault]
[Mercedes, Bmw] is NOT in [Fiat, Bmw, Audi, Porsche, Renault]
```

As you can see “containsAll” will return Boolean true only if all elements of the collection that we are passing as argument are contained in the set.

Checking if the set is empty:

```
listOf(
    setOf(),
    setOf(""),
    cars
).forEach {

    if (it.isEmpty()){
        println("Set: empty")
    } else {
        println("Set: $it")
    }
}
```

We have defined a list containing three members: one completely empty set, set with one member – empty string and cars set that we defined in our previous examples. We will check each and print message with proper text depending on the state of the set – it is or it is not empty. Let’s run our program:

```
Set: empty
Set: []
Set: [Fiat, Bmw, Audi, Porsche, Renault]
```

The first set was empty. Pay attention to the second row, we have printed one member, empty string, so this set is not empty after all! Finally, the last one has “visible” members.

Performing filtering:

```
cars.filter { it.length > 3 }.forEach {

    println("Car (name length > 3): $it")
}

cars.filterNot { it.length > 3 }.forEach {

    println("Car (name length <= 3): $it")
}
```

In this example, we have filtered all items from the set which have a size bigger than three and then, all that don't have a size bigger than three (they are less or equal to three). Let's run this and see what output is produced:

```
Car (name length > 3): Fiat
Car (name length > 3): Audi
Car (name length > 3): Porsche
Car (name length > 3): Renault
Car (name length <= 3): Bmw
```

Perform "+" and "-" operations:

```
val setA = setOf("Hello", "World", "Who")
val setB = setOf("Who", "Are", "You")

val setC = setA + setB
val setD = setC – setA
```

Let's see what do we get when we perform "+" and "-" operations on two sets:

```
println("$setA + $setB = $setC")
println("$setC + $setA = $setD")
```

Which will produce the following output:

```
[Hello, World, Who] + [Who, Are, You] =
   [Hello, World, Who, Are, You]

[Hello, World, Who, Are, You] + [Hello, World, Who] =
   [Are, You]
```

As you can see making a set that contains everything from two or more sets (addition) is very simple. The same applies to the subtraction operation.

Mutable collections

The main difference between immutable and mutable collections is that mutable collections can change. To illustrate this we will present a couple of examples for lists, maps, and sets.

The following code snippet will show us how to instantiate them:

```
val mutableList = mutableListOf(2, 4, 6)
val mutableList2 = mutableListOf("Some", "Word")
val mutableList3 = mutableListOf<String>()

val mutableMap = mutableMapOf("something" to 1, "else" to 2)
val mutableMap2 = mutableMapOf(
    Pair(1, "Plane"), Pair(2, "Car")
)

val mutableSet = mutableSetOf(2, 2, 3)
```

Let’s modify each collection by adding new items:

```
mutableList.add(3)
mutableList.addAll(listOf(3, 5, 7))

mutableList2.add(0, "New")
mutableList3.addAll(0, listOf("Hello", "World"))
mutableList3.add("Another")
mutableList3.add("One")

mutableMap["something"] = -1
mutableMap["new"] = 3
mutableMap2.putAll(mapOf(1 to "Train", 3 to "Boat"))

mutableSet.addAll(listOf(2, 2, 4))
mutableSet.add(5)
```

Since we have added new items into collections (and in some of these updated existing ones), let’s have a look at the content of each collection:

```
println(mutableList)
println(mutableList2)

println(mutableMap)
println(mutableMap2)

println(mutableSet)
```

Executing this snippet will reveal the following data:

```
[2, 4, 6, 3, 3, 5, 7]
[Hello, World, New, Some, Word, Another, One]
{something=-1, else=2, new=3}
{1=Train, 2=Car, 3=Boat}
[2, 3, 4, 5]
```

Finally, we will perform items removal:

```
mutableList.removeAt(0) // Remove element at position
mutableList2.remove("Word") // Remove object
mutableMap.remove("new") // Remove by map key
mutableSet.remove(4) // Remove object
mutableSet.removeAll(listOf(3, 5)) // Remove all objects
```

If we print out the content of collections that we have changed by removing items:

```
println(mutableList)
println(mutableList2)
println(mutableMap)
println(mutableSet)
```

We will have the following data:

```
[4, 6, 3, 3, 5, 7]
[Hello, World, New, Some, Another, One]
{something=-1, else=2}
[2]
```

As you can see the content of each collection that we have touched by proper remove function has been changed.

To remove all items from collection use “clear” function (like in the following source code snippet):

```
mutableList.clear()
mutableList2.clear()
mutableMap.clear()
mutableSet.clear()
```

If we print the content of each of these collections:

```
println(mutableList)
println(mutableList2)
println(mutableMap)
println(mutableSet)
```

Each of the collections will be empty:

```
[]
[]
{}
[]
```

Traversing

As you remember from previous sections we have traversed (iterated) through our collections. Let’s do one more example of traversing. We will traverse a list and a map. Open “Traverse.kt” from book code examples:

```
fun traverse(map: Map<*, *>) {
    map.forEach { (key, value) ->

        println("$key -> $value")
    }
}

fun traverse(list: List<Map<*, *>>) {
    list.forEach {

        traverse(it)
        if (list.indexOf(it) != list.lastIndex) {
            println("- - -")
        }
    }
}
```

Both ”traverse” functions iterate through the collection that has been passed as the function parameter. The second version of the “traverse” function traverses a map. For each keyset, it prints our values for the kay and for the value. The first version of the “traverse” function traverses through the list of maps and calls “traverse” that will traverse a map. Let’s define some collections (list and maps):

```
val maps = listOf(

    mapOf(1 to "First", 2 to "Second"),
    mapOf(
        "John" to "Smith",
        "John" to "Doe",
        "Lorem" to "Ipsum"), // <-- '"John" to "Doe"'
                        // overwrites '"John" to "Smith"'
                        // pair because of same key:
                        // "John"

    mapOf(1 to true, 2 to false, 3 to false, 4 to true)
)
```

If we execute the “traverse” function on the “maps” list:

```
traverse(maps)
```

The following program output will be produced:

```
1 -> First
2 -> Second
- - -
John -> Doe
Lorem -> Ipsum
- - -
1 -> true
2 -> false
3 -> false
4 -> true
```

Predicates

Predicates are very powerful features of the Kotlin programming language used in the filtering of collections. Simply said, predicates represent the lambda function used to filter collection items. We will demonstrate its usage on examples provided in ”Predicates.kt”. Let’s open it and go through examples step by step.

We will first define a function for some kind of check. For example, we will check if the passed number is positive. The function will be assigned to a variable. Check is illustrated by the code snippet:

```
val check: (Int) -> Boolean = { it > 0 }
```

Then, we will use ”check” to get what is important to us. Let’s assume that we will be interested if we have at least one positive number as a member of the list, or if we have all numbers positive or exact number of positive items. We would create something like this:

```
fun atLeastOnePositive(items: List<Int>): Boolean
{

   return items.any(check)
}

fun hasAllItemsPositive(items: List<Int>): Boolean
{

   return items.all(check)
}

fun numberOfPositiveNumbers(items: List<Int>): Int {

   return items.count(check)
}
```

To try out these three functions we will define simple list of numbers:

```
val numbers = listOf(-3, -2, -1, 0, 1, 2, 3, 4, 5)
```

Then, we will obtain results using them:

```
val atLeastOnePositive = atLeastOnePositive(numbers)
val hasAllItemsPositive = hasAllItemsPositive(numbers)
val numberOfPositiveNumbers = numberOfPositiveNumbers(numbers)
```

As you can see we have obtained information:

- If at least any list member has a positive value
- If the list has all positive values

- Exact number of positive values in the list.

Let’s see what are our results by printing them out:

```
println("Collection $numbers: ")
println("- has at least one positive: $atLeastOnePositive")
println("- has all items positive: $hasAllItemsPositive")
println("- number of positive numbers: $numberOfPositiveNumbers")
```

The following output will be produced:

```
Collection [-3, -2, -1, 0, 1, 2, 3, 4, 5]:
- has at least one positive: true
- has all items positive: false
- number of positive numbers: 5
```

Mapping

Mapping is used when we might have to do some operations to modify a collection according to our requirements using certain conditions. “map” function returns a list containing the results of applying a certain set of transformations to each element in the original collection. To better understand this we will show you a proper example. Open “Map.kt” from book code examples:

```
val numbers = listOf(0, 1, 2, 3, 4, 5)
val squares = numbers.map { it * it }
```

This is such an easy way to produce a list of squares! Let’s iterate through the list:

```
numbers.forEachIndexed { index, it ->

    println("Square of $it is: ${squares[index]}")
}
```

If we execute our program the following output will be produced:

```
Square of 0 is: 0
Square of 1 is: 1
Square of 2 is: 4
Square of 3 is: 9
Square of 4 is: 16
Square of 5 is: 25
```

Flattening

We will play now with Kotlin flat map. Kotlin's "flatMap" function returns a single list of all elements yielded from the results of the transform function being invoked on each element of the original collection. Open "FlatMap.kt" from book code examples and take a look at the first example:

```
val animals = mapOf(
   0 to "Elephant", 1 to "Lion", 2 to "Snake", 3 to "Ape"
)
val animalsList = animals.flatMap { listOf(it.value) }
```

As it is obvious we have defined a list that contains a map of Integer identifiers to the name of an animal. Then, we are mapping map values (animal names) into the list. If we print out the contents of the map and the list:

```
println(animals)
println(animalsList)
```

The following output will be generated:

```
{0=Elephant, 1=Lion, 2=Snake, 3=Ape}
[Elephant, Lion, Snake, Ape]
```

Let's have a look into another example of "flatMap" usage:

```
val words = listOf(

   "Hello", "World", "Airplane", "Car",
   "Beethoven", "Lorem", "Ipsum"
)

val allLetters = words.flatMap { it.toLowerCase().toList() }
```

We have created a list of some words. Then, we are mapping each word's letter into the "allLettersList". If we print it out:

```
println(allLetters)
```

The content of the “allLetters” list is the following:

```
[
    h, e, l, l, o, w, o, r, l, d, a, i, r, p, l, a, n, e, c,
    a, r, b, e, e, t, h, o, v, e, n, l, o, r, e, m, i, p, s,
    u, m
]
```

As you can see we have created a list that contains only letters used in these words. However, there is a lot of duplicates in the list which is annoying. To get a rid of duplicates we will sort the list (because we want to have all letters sorted in alphabetical order) and add all members into the set which will remove our duplicates:

```
val letters = mutableSetOf<Char>()
val sortedLetters = mutableListOf<Char>()

letters.addAll(allLetters)
sortedLetters.addAll(letters)
sortedLetters.sort()
```

If we print out the content of the set:

```
println(sortedLetters)
```

The following output will be produced:

```
[a, b, c, d, e, h, i, l, m, n, o, p, r, s, t, u, v, w]
```

Another more easy way to get rid of duplicates is to use “distinct” function:

```
val sortedLetters = allLetters.distinct().sorted()
```

Which will produce exactly the same results.

Combining “map” and “flatMap”

In the following example, we will combine “map” and “flatMap”. From book code examples open “Mapping.kt”. The example illustrates how we can combine both functions in other to achieve the desired goal:

```
class Vehicle(val name: String)
```

```
val cars = listOf(

    Vehicle("Bmw"),
    Vehicle("Mercedes"),
    Vehicle("Toyota")
)

val busses = listOf(

    Vehicle("Solaris"),
    Vehicle("Champion"),
    Vehicle("Ikarbus")
)

val vehicles = listOf(cars, busses)
val manufacturers = vehicles.flatMap {
    it
}.map {
    it.name
}
```

Last part can be simplified:

```
val manufacturers = vehicles.flatten().map {
    it.name
}
```

From all collections, in the given collection the “flatten” function returns a single list of all elements.

We will go step by step through the example. First, we defined a simple class to represent a vehicle entity. The only attribute of the vehicle is its name.

Then, we have created two lists. The first list contains car vehicles, and the second one contains bus vehicles. Since we are interested in creating a unique list of all vehicle names we are putting all vehicles under one hood, the “vehicles” list (the list that contains the vehicle list).

The last step is mapping everything into a “manufacturers” list that will contain only names. To achieve this we have used “map” and “flatMap” (“flatten”) functions. Let’s print out the content of the final list:

```
println(manufacturers)
```

Which will produce the following content:

```
[Bmw, Mercedes, Toyota, Solaris, Champion, Ikarbus]
```

Finally, we can sort our list:

```
println(manufacturers.sorted())
```

And have the following output of program execution:

```
[Bmw, Champion, Ikarbus, Mercedes, Solaris, Toyota]
```

Finding maximum and minimum

If we need to determine maximal or minimal value of some list “maxOrNull” and “minOrNull” functions are used. From book code examples open ”MaxMin.kt” and take a look at the lines:

```
val items = listOf(2, 4, 6, 8, 10)
val max = items.maxOrNull()
val min = items.minOrNull()
```

As you can see we have obtained maximal and minimal values and it is very simple. Max function returns the largest element or null (if there are no elements in the list). Min returns the smallest element (or null if there are no elements in the list).

Let’s see what are our results:

```
println("For: $items")
println("Max. is: $max")
println("Min. is: $min")
```

Which will produce the following results output:

```
For: [2, 4, 6, 8, 10]
Max. is: 10
Min. is: 2
```

Sorting

Sorting is a common need for every developer. In Kotlin, performing sorting operation on collections is really easy. Open “Sort.kt” from book code examples:

```
val numbers = listOf(1, 8, 9, 4, 5, 22, 44, 645, 67).sorted()
println("Sorted numbers: $numbers")

val stringNumbers = listOf("1", "-3", "5")
                    .sortedBy { it.toInt() }

println("Sorted string numbers: $stringNumbers")

val words = mapOf("Yin" to "Yang", "Lorem" to "Ipsum",
   "Hello" to "World").toSortedMap()

println("Sorted map: $words")

val set = setOf(2, 4, 4, 2, 0, 1, 3, 6, 4, 5).sorted()
println("Sorted set: $set")

val resorted = set.sortedBy { it % 2 == 0 }
println("Re-sorted set: $resorted")
```

To sort these collections we have used two functions:

- “sorted”: returns a list of all elements that are sorted according to their natural order
- “sortedBy”: returns a list of all elements that are sorted according to the natural order of the value returned by the provided selector function.

Executing our little program will reveal the following results:

```
Sorted numbers: [1, 4, 5, 8, 9, 22, 44, 67, 645]
Sorted string numbers: [-3, 1, 5]
Sorted map: {Hello=World, Lorem=Ipsum, Yin=Yang}
Sorted set: [0, 1, 2, 3, 4, 5, 6]
Re-sorted set: [1, 3, 5, 0, 2, 4, 6]
```

Sum

Another easy operation that can be performed on collections is ”sum”. From book code examples open “Sum.kt” and take a look at the example:

```
val itemsToSum = listOf(1, 3, 5)
val sum = itemsToSum.sum()
```

The “sum” function returns the sum of all elements in the collection. Let’s print out our sum:

```
println("Sum of $itemsToSum is: $sum")
```

Which will produce the following output:

```
Sum of [1, 3, 5] is: 9
```

Grouping

The “groupBy” function groups elements of the original collection by the key returned by the given ”keySelector” function applied to each element and returns a map where each group key is associated with a list of corresponding elements.

The resulting map preserves the entry iteration order of the keys created from the original collection. To illustrate this we have created a simple example. Open “GroupBy.kt” from book code examples and take a look at lines:

```
val words = listOf("a", "plane", "to", "car", "window")
val grouped = words.groupBy { it.length }
```

If we print out our results:

```
println(words)
println(grouped)
```

The following output will be produced:

```
[a, plane, to, car, window]
{1=[a], 5=[plane], 2=[to], 3=[car], 6=[window]}
```

Partitioning

“partition” function splits the original collection into pair of lists. First list contains elements for which predicate is true. Second list contains elements for which predicate is false. We demonstrate use of “partition” function in “Partition.kt” from book code examples example:

```
val numbers = listOf(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
val partitioned = numbers.partition { it % 2 == 0 }
```

“partitioned” constant is instance of ”Pair” data type. Let’s print out the content of original collection and partitioned data:

```
println("Original: $numbers")
println("Partitioned: $partitioned")
```

Which will produce the following output:

```
Original: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Partitioned: ([0, 2, 4, 6, 8, 10], [1, 3, 5, 7, 9])
```

As you can see we have two lists as members of the Pair. Members of the first list are the numbers who satisfied the condition with boolean True. The others are on the second list.

Folding

“fold” function groups elements in a way that the grouping is performed from the grouping source by key and applies the operation to the elements of each group sequentially. It passes the previously accumulated value and the current element as argument, and stores the results in a newly created map.

From book code examples open ”Fold.kt”:

```
val numbers = listOf(2, 4, 6, 8, 10, -20)
val folded = numbers.fold(

    initial = 0,
    operation = { part, element -> part + element }
)
```

The folded result will be an integer with a value of exactly ten. Let’s check this:

```
println(numbers)
println(folded)
```

The following output is produced:

```
[2, 4, 6, 8, 10, -20]
10
```

If you still do not understand how these works take a closer look into he source code of the “fold” function:

```
public inline fun <T, R> Iterable<T>.fold(

   initial: R,
   operation: (acc: R, T) -> R
): R {

   var accumulator = initial
   for (element in this) accumulator =
      operation(accumulator, element)

   return accumulator
}
```

This example uses an inline generic data type. We will talk about generics in upcoming sections of this book.

References

In this section, we will talk about references. We will explain the purpose of references and guide you through each type of them.

References represent data types that contain addresses (references) of dynamically created objects. In Kotlin (and Java) there are four different kinds of references based on the way how the data is garbage collected:

- Strong references
- Weak references
- Soft references
- Phantom references
- Atomic references (they are not actually in this group, however, we will cover them because they provide atomic access to wrapped objects).

Strong references

Strong references are the default type of reference object. An object that has active strong reference can’t be garbage collected by a JVM garbage collector. This can happen in situation when the variable that is strongly referenced points to null. Let’s have a look at the example. From book code example open “Strong.kt”:

```
data class Wrapper(val what: Any)

val number = 1
val wrapper = Wrapper(100)

var number2: Int? = null
var wrapper2: Wrapper? = null
```

“number”, “number2”, “wrapper” and “wrapper2” are all strong references. However, “number2” and “wrapper2” have null values.

Working with strong references can sometimes lead to memory leaks. A memory leak is a situation when there are objects present in the heap that are no longer used, but the garbage collector is unable to remove them from memory. Because of this, they are unnecessarily maintained.

A memory leaks block and consumes memory resources. This degrades system performance over time and can lead to slack performance or crash of the application (program). If it is not dealt with, the application will at some point exhaust all resources. Finally, the program will terminate with a fatal ”OutOfMemoryError”.

Types of objects that reside in heap memory are:

- Referenced objects, who have still active references within the program
- Unreferenced objects, who don't have any active references.

The garbage collector removes unreferenced objects periodically. Unfortunately, the garbage collector never collects the objects that are still being referenced.

Memory leaks

As we mentioned in the previous section, sometimes we can create memory leaks. In this section, we will provide some examples of memory leaks. Memory leaks can happen in various ways. Some common memory leaks are:

- Memory leaks through static fields
- Memory leaks through unclosed resources
- Memory leaks through improper ”equals” and “hashCode” functions implementations
- Memory leaks through nested classes that reference outer class.

There are more different ways for memory leaks to happen. Let’s give example for each of these cases. We will start with **memory leaks that are introduced through static (global) fields**. Anything static (and global scope) has a life that usually matches the entire lifetime of the running program. From book code examples open “StaticFieldLeak.kt” and take a look at the code:

```
val staticList = mutableListOf<Double>()

fun main() {

    fun populate() {

        for (x in 0 until 100 * 1000 * 1000) {
            staticList.add(x.toDouble())
        }
    }

    fun printMemoryStats() {

        val heapSize = Runtime.getRuntime().totalMemory()
        println("Heap size: $heapSize")
    }

    printMemoryStats()

    println("Populating static list")
    populate()

    printMemoryStats()
    println("Executing garbage collection")
    System.gc()

    printMemoryStats()
}
```

What this example is illustrating is the following:

- We have created a global mutable list named "staticList"
- We are printing the current size of the heap memory (before the list is filled with doubles)
- After the list is filled with objects we are printing again heap memory size
- Then, we are triggering garbage collection explicitly by calling the "gc" function from runtime. Calling this function suggests that the JVM expends effort toward recycling unused objects to make the memory they currently occupy available for quick reuse. When control returns from the function call, the virtual machine has made its best effort to recycle all discarded objects. The virtual machine performs this recycling process automatically as needed, in a separate thread, even if the "gc" function is not invoked explicitly. The "System.gc()" is the standard way of invoking this function.
- Finally, we are printing again the size of the heap memory.

As you will see the heap size is increasing through the execution of our program. Once the program is executed something similar to this will be generated as the final output:

```
Heap size: 257425408
Populating static list
Heap size: 3669491712
Executing garbage collection
Heap size: 3669491712
```

Memory leaks that are introduced through unclosed resources:

When opening a new connection (for example to a database) or a new stream causes new memory allocation. If we forget to close any of these resources, this can block the memory. For example, in case of an exception that prevents the program execution from reaching the statement that's handling the code to close these resources.

To be sure that we don't get memory leaks caused by unclosed connections or streams, always use the "finally" block to close the program's resources.

Pay attention that the code that closes the resources should not itself have any exceptions. That is meant for the “finally” block too.

Memory leaks that are introduced through improper ”equals” and “hashCode” functions implementations:

Not writing properly overridden functions for ”equals” and “hashCode” functions can be a source of memory leakage. “HashSet” and ”HashMap” data types use these functions. If they're not overridden properly, then they can become a source for potential memory leak problems.

Let’s start with defining a simple class with one field. From book code examples open “OverrideLeak.kt”:

```
class Wrapper(var data: Int)
```

The next thing that we will do is that we will insert duplicate “Vehicle” objects into a map that uses this key. As you probably remember, the map cannot contain duplicate keys.

```
val word = "Kotlin"
val instances = mutableMapOf<Wrapper, Int>()
for (x in 0 until 100) {

    instances[Wrapper(word)] = x
}
```

In this case, we are using “Wrapper” as the key. Because the map doesn't allow duplicated keys, the numerous duplicate objects that we have inserted as the key should not increase the memory. However, **this does not happen!**

Because we didn’t define proper “equals” function implementation, the duplicate objects will pile up and increase the memory consumption. But, if we did overridde the “equals” and “hashCode” functions properly, there would be only one “Wrapper” class instance in the map.

Let's take a look at the proper implementations of these functions in the “Wrapper” class:

```
class Wrapper(var data: String) {
    override fun equals(other: Any?): Boolean {
        if (other === this) {

            return true
        }
        if (other !is Wrapper) {

            return false
        }
        return other.data == data
    }

    override fun hashCode(): Int {

        return Objects.hash(data)
    }
}
```

If we run this program not, there will be only one instance of the “Wrapper” class in our map. Let’s check this by printing out the size of the map:

```
println(

    "Number of ${Wrapper::class.simpleName} instances is:
    ${instances.size}"
)
```

Which will produce the following output:

```
Number of Wrapper instances is: 1
```

Memory leaks that are introduced through nested classes that reference outer classes:

This is a common problem in classic Java. If we are using anonymous classes (non-static nested classes), for its initialization, these classes **always require an instance of the enclosing class**. Every non-static nested class has, by default, an implicit reference to its container (its containing class). So how this can cause memory leakage? If we use this nested class instance in our application, then even after our container class instance goes out of scope, **it will not be garbage collected**.

Fortunately, if we just declare the nested class as static our problem is solved.

Both Java and Kotlin support nested classes, but some important differences are worth keeping in mind. As we have illustrated in the explanation of this memory leakage type, in Java, nested classes declare (implicitly) a reference to the surrounding class. On the other hand, nested classes in Kotlin **do not declare any implicit reference to surrounding classes**. So they could be considered safer by default. By removing this reference, we get rid of possible memory leaks and other possible issues.

In Kotlin, if we need to access the external class, we have to apply the ”inner” modifier in the nested class declaration. Let’s illustrate this. From book code examples open “InnerClassLeak.kt”:

```
class Wrapper(private var data: String) {

    inner class Inner {

        fun print() {

            // We are accessing to outer
            // class reference and its field:
            println("Data: $data")
        }
    }
}
```

Nested classes in both programming languages have the same semantics, but they behave differently. In Java, nested classes have access by default to external classes. On the contrary, Kotlin classes do not have it, so we apply the ”inner” modifier to get the “default Java behavior” (and create in our program a potential memory leak).

Weak references

A weak reference is a reference made that is not strong enough to make the object (instance) remain in memory. So, weak references can let the garbage collector decide an object’s reachability and whether the object in question should be kept in memory or not.

Weak references need to be declared explicitly as by default Kotlin marks a reference as a strong (hard) reference. It means that an object has neither strong nor soft references pointing to it and can only be reached by traversing through a weak reference.

If the object is weakly referenced then the garbage collector removes it from memory which clears up more space and makes for better memory management.

After the garbage collector has removed the weak reference, the reference is placed in a reference queue, and the formerly weak-reachable objects are finalized.

If the garbage collector determines that an object is weakly reachable. At that time it will atomically clear all weak references to that object and all weak references to any other weakly-reachable objects from which that object is reachable through a chain of strong (hard) and soft references.

Let’s have a look at the example that illustrates the use of weak references. From book code examples open “Weak.kt”:

```
// Referent definition, some 'dummy' class:
class Dummy {

    // Referent's function:
    fun hello() = println("Dummy: ${hashCode()}")
}

// Referent:
val dummy = Dummy()

// We are initializing weak reference
// by passing a referent as a parameter:
val weak = WeakReference(dummy)

weak.get()?.hello()

// Release weak reference:
weak.clear()

weak.get()?.hello()
```

As you can see we have created a weak reference by calling the “WeakReference” class constructor with referent as the argument. A new weak reference that refers to the given object is created. A referent is an object to which a new weak reference will refer.

Obtaining access to referent is achieved via the “get” function. The function returns this reference object's referent. This function returns null if this reference object has been cleared, either by the program or by the garbage collector.

Finally, we are calling “clear”. “clear” function clears this reference object. Invoking this function will not cause this object to be enqueued. This function is invoked only by our code. However, when the garbage collector clears references it does so directly, without invoking this function.

Let’s run our program:

```
Dummy: 491044090
```

As you can see, the “hello” function has been executed only once since for the second execution there was no referent pointed by our weak reference.

Soft references

A soft reference object can be cleared by the Garbage Collector in response to a a memory demand. When a Garbage Collector is called, it iterates over all elements in the heap. GC keeps reference-type objects in a special type of queue. GC determines which instances should be removed by removing objects from that queue after all objects in the heap have been checked. It is guaranteed to be cleared soft references to softly-reachable objects before a JVM throws an OutOfMemoryError.

Because of this, the time when a soft reference gets cleared is not guaranteed or the order in which a set of such references to different objects gets cleared.

To demonstrate the practical use of soft references through the code open ”Soft.kt” from book code examples:

```
class Dummy {

    fun hello() = println("Dummy: ${hashCode()}")
}

val dummy = Dummy()
```

```
// We are initializing soft reference
// by passing a referent as a parameter
// exactly as we did with weak reference:
val soft = SoftReference(dummy)

soft.get()?.hello()

// Release soft reference:
soft.clear()

soft.get()?.hello()
```

Phantom references

Phantom references are rarely used in everyday development. However, it is worth mentioning that phantom references have two major differences compared to weak and soft references:

- A referent of a phantom reference cannot be obtained. The referent is never accessible directly through the API. Because of this, we need to work with a “ReferenceQueue” class.

- The Garbage Collector adds a phantom reference to a reference queue after the finalize function of its referent is executed. It implies that the instance is still in the memory.

References summary

Listing the reference types from the strongest to the weakest, the different levels of reachability reflect the life cycle of an object. They are defined as follows:

- An object is strongly reachable if it can be reached by some thread without traversing any reference objects. A newly created object is strongly reachable by the thread that has created it.

- An object is softly reachable if it is not strongly reachable but can be reached by traversing a soft reference.

- An object is weakly reachable if it is neither strongly nor softly reachable. It can be reached by traversing a weak reference. The object becomes eligible for finalization when the weak references to a weakly-reachable object are cleared.

- An object is phantom reachable if it is neither strongly, softly, nor weakly reachable. The object is phantom reachable if it has been finalized, and some phantom reference refers to it.

Finally, when the object is not reachable in any of the above ways it is unreachable, and therefore eligible for reclamation.

Atomic references

The atomic reference provides operations on an underlying object reference that can be read and written thread-safely. Thread-safe code only manipulates shared data structures in a manner that ensures that all threads behave properly and fulfill their design specifications without unintended interaction. Atomic reference contains advanced atomic (concurrent) operations. The ones that you will use mostly are "get" and "set". "get" and "set" functions have a purpose of reading and setting of the instance. Atomic reference should be used in situations where you need to do simple atomic (thread-safe) operations.

Let's have a look at the example of "AtomicReference" use in code. From book code examples open "Atomic.kt":

```
// We define some stub class:
class Dummy

val d1 = Dummy()
val d2 = Dummy()

// Atomic wrapper around the Integer:
val counter = AtomicInteger()

// Initially it does not point to any instance:
val reference = AtomicReference<Dummy>()

val t1 = Thread {

    println("Time 1: ${System.nanoTime()}")
    // We are setting new reference:
    reference.set(d1)
    println("Is d1: ${d1 === reference.get()}")
    println("Is d2: ${d2 === reference.get()}")

    // Increment the counter:
    counter.incrementAndGet()
}

val t2 = Thread {

    println("Time 2: ${System.nanoTime()}")
    reference.set(d2)
    println("Is d1: ${d1 === reference.get()}")
    println("Is d2: ${d2 === reference.get()}")
    counter.incrementAndGet()
}

// We will use executor which will ensure that
// t1 completes before t2:
val executor = Executors.newSingleThreadExecutor()

// We are sending threads to be executed sequentially:
executor.execute(t1)
executor.execute(t2)

// We wait until both threads complete
while (counter.get() < 2) {

    Thread.yield()
}

// Get the referent:
val instance = reference.get()

println("Time 3: ${System.nanoTime()}")
// Compare by reference:
println("Is d1: ${d1 === instance}")
println("Is d2: ${d2 === instance}")

// Clear the reference:
reference.set(null)
exitProcess(0)
```

Follow the source code comments to better understand every step of this simple example. Executing this program will give us output similar to this one:

```
Time 1: 5197247991885
Is d1: true
Is d2: false

Time 2: 5197248336431
Is d1: false
Is d2: true

Time 3: 5197258452231
Is d1: false
Is d2: true
```

This reference

"this" is a reference variable that refers to the current object. From book code examples open "This.kt":

```
class Wrapper(private val value: Int) {

    fun sum(value: Int) {

        val sum = this.value + value
        println("Sum: $sum")
    }
}

val wrapper = Wrapper(1)
wrapper.sum(3)
```

In this example, the crucial part is the "sum" function. To differentiate between function's argument named "value" and class constant with the same name we have used "this" reference. Thanks to that, we are accessing the current object and all current object members. In this case, "value" is constant.

Executing the program will produce the following output:

```
Sum: 4
```

Generic data types

“Generic” data type means that the type is parameterized. The core value of “generic” data types is to allow types such as ”Integer”, ”Float”, “String” and so on, and user-defined types as well to be parameters to functions, objects, classes, and interfaces. By using “generic” data types, it is possible to create classes that work with different data types.

Let's take a look at the example that shows us the simple use of ”generics” in Kotlin. From book code examples open “Generics.kt”:

```
class Container<T>(val data: T) {

    fun describe() {
        println("Data contained: $data")
    }
}
```

We have defined a class for “generic” data type of some “T”. That means that we can instantiate the “Container” class and have a “data” field of any type. The class contains a “describe” function that will print-line string representation of the “data” property. Let’s try it out:

```
val data1 = Container(1)
val data2 = Container("Hello world!")
val data3 = Container(true)
```

Executing our program will produce the following result:

```
Data contained: 1
Data contained: Hello world!
Data contained: true
```

Ok, here we demonstrated basic usage of generics. Let’s introduce more precision into the class types. From book code examples open the “Cars.kt” source code file:

```
abstract class Car(var model: String) {

    abstract var name: String
}
```

```
class Bmw(model: String) : Car(model) {

    override var name: String = "Bmw"
}

class Mercedes(model: String) : Car(model) {

    override var name: String = "Mercedes"
}
```

“Car”, “Bmw” and “Mercedes” are just simple classes. Now, we will define a class for car washing that only deals with “Car” types:

```
class CarWash<in C : Car> {

    fun washCar(car: C) {

        println("Washing ${car.name} ${car.model}")
    }
}
```

“CarWash” class works with generic data types that inherit super-type “Car”. Also, using “in” keyword tells us that we will only consume our generic type “C”. We will not produce it as a result of our class operations.

On the other hand, we can define a class that only produces this data type:

```
abstract class CarFactory<out C : Car> {

    abstract fun forgeCar(model: String): C
}
```

For each type of car, we will create a separate factory. One factory for Bmws, one for Mercedeses:

```
class BmwFactory : CarFactory<Bmw>() {

    override fun forgeCar(model: String): Bmw {
        return Bmw(model)
    }
}
```

```
class MercedesFactory : CarFactory<Mercedes>() {

    override fun forgeCar(model: String): Mercedes {
        return Mercedes(model)
    }
}
```

Let’s try this out:

```
val bmwFactory = BmwFactory()
val mercedesFactory = MercedesFactory()

val bmw = bmwFactory.forgeCar("M4")
val mercedes = mercedesFactory.forgeCar("AMG GT")

val bmwCarWasher: CarWash<Bmw> = CarWash()
val mercedesCarWasher: CarWash<Mercedes> = CarWash()
val generalCarWasher: CarWash<Car> = CarWash()

bmwCarWasher.washCar(bmw)
mercedesCarWasher.washCar(mercedes)
generalCarWasher.washCar(bmw)
generalCarWasher.washCar(mercedes)
```

If we run this and we “wash” our cars, the following output will be produced by the program:

```
Washing Bmw M4
Washing Mercedes AMG GT
Washing Bmw M4
Washing Mercedes AMG GT
```

Generic functions

“Generic functions” are functions capable of working with any data type (“generic”) or subtypes of some super-type. We will illustrate this Kotlin feature through a simple example. From book code examples open “GenericFunctions.kt”:

```
abstract class Engine {
    abstract val power: Long

    override fun toString(): String {

        return "Engine(power=$power)"
    }
}

class RocketEngine : Engine() {

    override val power: Long
        get() = 1000
}

class TruckEngine : Engine() {

    override val power: Long
        get() = 100
}
```

We have defined a few simple classes, basic abstraction with final implementations: “Engine”, “RocketEngine”, and “TruckEngine”.

Then:

```
class EngineDiagnostics {

    fun <T : Engine> checkEngine(engine: T) {

        println(engine)
    }
}
```

We have defined another class that contains a “generic function” capable of dealing with “engines”. “checkEngine” is “generic” and it is capable of work with all classes that inherit the “Engine” abstract class. That is any data type “T” which inherits “Engine” abstraction.

Let’s try it out:

```
val truckEngine = TruckEngine()
val rocketEngine = RocketEngine()
val diagnostics = EngineDiagnostics()

diagnostics.checkEngine(truckEngine)
diagnostics.checkEngine(rocketEngine)
```

Running “engine checks” on “Engine” instances will produce the following result:

```
Engine(power=100)
Engine(power=1000)
```

Generics wildcards

Generics wildcards are used to indicate that input or output data types are subtypes of some super-type, which means that some type that we use must inherit some other super-type. Let’s have a look at the example that illustrates how we can do this. From book code examples open “Extends.kt”:

```
abstract class Airplane {

   abstract fun describe()
}

class Boeing : Airplane() {

   override fun describe() {
      println("Being")
   }
}

class Airbus : Airplane() {

   override fun describe() {
      println("Airbus")
   }
}

class AirplaneDescriptor {

   fun <T : Airplane> describe(airplane: T) {
      airplane.describe()
   }
}
```

We have defined a couple of classes. “Airplane” class is the root abstraction class for airplanes. “Boeing” and “Airbus” are the implementation classes. For us, the key class of this example is the “AirplaneDescriptor” class. The “AirplaneDescriptor” class has only one function that works with generic data type T which inherits the “Airplane” abstraction. This example, “describes” all airplanes. If we try to pass any other data type to the “describe” function compiler will complain. Let’s try this:

```
val airplane1 = Airbus()
val airplane2 = Boeing()
val descriptor = AirplaneDescriptor()

descriptor.describe(airplane1)
descriptor.describe(airplane2)
```

Executing the program will produce the following result:

```
Airbus
Being
```

Enumeration

You will sometimes need the data type that can have only certain values. For this purpose enumeration was introduced. “Enumeration” represents a named list of constants. In Kotlin “enum” has its specialized data type which tells us that something has several possible values.

It is important to note that Kotlin’s “enums” are classes. Because of that, “enums” can have properties, functions, etc. Each of the “enum” constants acts as a separate instance of the class which cannot be created by the “constructor” function. These instances are separated by commas.

Let's have a look at some “enums” in Kotlin. From book code examples open “Enums.kt”:

```
enum class TIME {

    MICROSECOND,
    MILLISECOND,
    SECOND,
    MINUTE,
    HOUR,
    DAY,
    WEEK,
    MONTH
}

enum class COUNTRY(val capital: String) {

    ITALY("Rome"),
    SPAIN("Madrid"),
    RUSSIA("Moscow"),
    SCOTLAND("Edinburgh")
}
```

We have defined two “enums”. The first enum “TIME” does not have any values associated with its constants. On the other hand, our second example, “COUNTRY” has the associated value.

This is how we could access them:

```
println("Time unit: ${TIME.MILLISECOND}")
println("Time unit: ${TIME.SECOND}")
println("Time unit: ${TIME.MINUTE}")

println(
    "Country vs capital: ${COUNTRY.ITALY} ->
        ${COUNTRY.ITALY.capital}"
)
println(
    "Country vs capital: ${COUNTRY.SPAIN} ->
        ${COUNTRY.SPAIN.capital}"
)
println(
    "Country vs capital: ${COUNTRY.RUSSIA} ->
        ${COUNTRY.RUSSIA.capital}"
)
```

Executing this simple code snippet will produce us the following output:

```
Time unit: MILLISECOND
Time unit: SECOND
Time unit: MINUTE

Country vs capital: ITALY -> Rome
Country vs capital: SPAIN -> Madrid
Country vs capital: RUSSIA -> Moscow
```

Let's take a look at the example in which “enums” can have function implementations. From book code examples open “EnumsExtended.kt”:

```
enum class CAPABILITIES {

    POWER_ON {
        override fun execute() {
            println("Powering on")
        }
    },
    POWER_OFF {
        override fun execute() {
            println("Powering off")
        }
    },
    MUTE {
        override fun execute() {
            println("Muting audio")
        }
    },
    VOLUME_UP {
        override fun execute() {
            println("Vol +")
        }
    },
    VOLUME_DOWN {
        override fun execute() {
            println("Vol -")
        }
    };

    abstract fun execute()
}
```

We have defined “enum” called “CAPABILITIES” with abstract function member: “execute”. Because of this, each “enum” constant must implement it. Let’s try it:

```
val capabilities = listOf(

    CAPABILITIES.POWER_ON,
    CAPABILITIES.VOLUME_UP,
    CAPABILITIES.VOLUME_UP,
    CAPABILITIES.VOLUME_DOWN,
    CAPABILITIES.POWER_OFF
)

for (capability in capabilities) {
    capability.execute()
}
```

Running this little “enums” program will produce the following output:

```
Powering on
Vol +
Vol +
Vol -
Powering off
```

Our last “enums” example will demonstrate the use of two important “enum” functions. From book code examples open “EnumConstants.kt”:

```
enum class PLANETS {

    EARTH,
    MARS,
    PLUTO
}
```

We start this example by defining simple enum. Then, we try these functions:

```
// Returns PLUTO enum item:
println("Value of: ${PLANETS.valueOf("PLUTO")}")

try {

    val planetBlah = PLANETS.valueOf("BLAH")
    println("Planet: $planetBlah")
} catch (e: IllegalArgumentException) {

    println(e.message)
}

// values() func. returns array containing all enum items:
println("Planets count: ${PLANETS.values().size}")
```

As you can see “valueOf” function returns the “enum” constant by the name. If there is no such constant defined, “IllegalArgumentException” is thrown. Lastly, if we ever need an array that contains all of the “enum” constants, we can use the “values” function.

If we run our code example the following output will be produced:

```
Value of: PLUTO

No enum constant net.milosvasic.fundamental.kotlin.object_oriented.PLANETS.BLAH

Planets count: 3
```

Sealed classes

For representing restricted class hierarchies sealed classes are used. Sealed class value can have one of the types defined within a limited set. The set of values for an enum type is also restricted, but each enum constant exists only as a single instance, whereas a subclass of a sealed class can have multiple instances which can contain a state.

A sealed class can have subclasses. All of them must be nested inside the declaration of the sealed class.

Note that classes that extend subclasses of a sealed class (indirect inheritors) can be placed anywhere, not necessarily inside the declaration of the sealed class.

One of the most frequent use cases for the sealed classes is using them in the “when” expression. The “else” clause is not needed if it’s possible to verify that the statement covers all cases.

From book code examples open “Sealed.kt”:

```
sealed class Specie {

    class Human(val race: String) : Specie()

    class Animal(
        val specie: String, val legsCount: Int
    ) : Specie()

    object Bacteria : Specie()
}
```

We have defined a simple “sealed class” to describe nature’s species. Species may be “human”, “animal” or like in our example “bacteria” which is an object.

Now we will define a simple function that will work with “species”:

```
fun describe(specie: Specie) = when (specie) {

    is Specie.Human -> "Human ${specie.race}"
    is Specie.Animal ->
"${specie.specie}, it has ${specie.legsCount} legs."

    is Specie.Bacteria -> "Some micro organism..."
}
```

Depending of the “specie” type it will return proper description string. Let’s try it:

```
val indian = describe(Specie.Human("Indian"))
val asian = describe(Specie.Human("Asian"))

println(indian)
println(asian)

val monkey = describe(Specie.Animal("Monkey", 2))
val horse = describe(Specie.Animal("Horse", 4))

println(monkey)
println(horse)

val bacteria = describe(Specie.Bacteria)
```

```
println(bacteria)
```

Running this small simple example will produce the following output:

```
Human Indian
Human Asian
Monkey, it has 2 legs.
Horse, it has 4 legs.
Some micro organism...
```

Annotations

Annotations represent a feature for attaching metadata to the code. To declare an annotation it is required to put the ”annotation” modifier in front of a class.

Annotation can have additional attributes. They can be specified by annotating the annotation class with meta-annotations such as:

- “@Target”:

This meta-annotation indicates the kinds of code elements that are possible targets of an annotation. If the target meta-annotation is not present on an annotation declaration, the annotation is applied to the following elements:

- CLASS,
- PROPERTY,
- FIELD,
- LOCAL_VARIABLE,
- VALUE_PARAMETER,
- CONSTRUCTOR,
- FUNCTION,
- PROPERTY_GETTER,
- PROPERTY_SETTER.

- “@Retention”:

Determines whether an annotation is stored in binary output and visible for reflection. By default, both are true.

- “@Repeatable”:

Determines that an annotation is applicable twice or more on a single code element.

- “@MustBeDocumented”:

Determines that an annotation is a part of public API. Therefore it should be included in the generated documentation for the element to which the annotation is applied.

We will illustrate use of annotations on the example from book code examples “Annotations.kt”:

```
@Target(
    AnnotationTarget.CLASS,
    AnnotationTarget.CONSTRUCTOR,
    AnnotationTarget.FUNCTION,
    AnnotationTarget.FIELD,
    AnnotationTarget.TYPE_PARAMETER,
    AnnotationTarget.VALUE_PARAMETER,
    AnnotationTarget.EXPRESSION,
    AnnotationTarget.PROPERTY_SETTER,
    AnnotationTarget.PROPERTY_GETTER
)
@Retention(AnnotationRetention.SOURCE)
@MustBeDocumented
annotation class Marker
```

And now we can use it to annotate our classes:

```
@Marker class Annotated {

    @Marker fun hello(@Marker who: String): String {

        return (@Marker "Hello $who")
    }
}
```

Annotating the primary constructor of the class is achievable by adding the constructor keyword to the constructor declaration and by adding the annotations before it. Like in this example:

```
class Annotated2 @Marker constructor()
```

To be able to apply the annotation to the constructor it is required that "AnnotationTarget.CONSTRUCTOR" is present in annotation's "@Target" definition.

Property accessors can be also annotated:

```
class Annotated3 {

    var number: Int? = null
        @Marker set
}
```

To be able to apply the annotation to the property accessors it is required that

"AnnotationTarget.PROPERTY_SETTER"
(or "AnnotationTarget.PROPERTY_GETTER")

is present in annotation's "@Target" definition.

Annotation constructors

Annotations can have parameterized constructors. Let's illustrate this with a simple example. From book code examples open "AnnotationConstructors.kt" and take a look:

```
annotation class Description(val desc: String)

@Description("I am a dummy class") class Dummy
```

We have defined "Description" annotation which accepts one string argument in its constructor. Then, we have used our annotation to "describe" the "Dummy" class.

Annotation constructors can have the following types of parameters:

- Other annotations
- Number data types: "Int", "Long" and others
- Strings
- Classes, for example, "Dummy::class"
- Enums
- Arrays of any of these data types.

Note:

Parameters cannot have nullable types. JVM does not support this.

Note:

The name of the annotation should not be prefixed with the "@" character if the annotation constructor parameter has the type of another annotation.

Lambdas in annotations

Lambdas can be annotated too. Annotation is applied to the "invoke" function. "invoke" function is the function into which the body of the lambda is generated. We will illustrate this with a simple example. From book code examples open "AnnotatingLambdas.kt":

```
val lambda = @Marker {

    println("Do something ...")
}
```

As you can see it is really simple to annotate the lambda function.

Arrays in annotations

As we have already mentioned, arrays can be used as the annotation parameters. Let's illustrate this with a simple example. From book code examples open "AnnotationArrays.kt":

```
annotation class Meta(val data: Array<String>)

@Meta(["Hello", "World"]) class Data
```

As you can see here, the "Meta" annotation accepts the array of strings as the argument. The "Data" class is annotated with "Meta" annotation. In this case array with two strings has been passed as an argument to the annotation.

Most frequently used annotations

In this section, we will mention a few of the most frequently used annotations and their purpose in everyday development. As most of them come directly from Java it is important to note that Java annotations are completely compatible with Kotlin.

Some of them are:

- ”SuppressWarnings”: used to suppress warnings issued by the compiler

- “Deprecated”: used to mark that the function or class are deprecated. Compiler prints warning for all marked classes and functions. Its main purpose is to inform the developer that marked function or class may be removed in future versions of the product.

- “Generated”: used to mark source code that has been generated. It can be specified on a class, function, or field. It can also be used to differentiate user-written code from generated code in a single file.

- ”Resource”: used to declare a reference to a resource. It can be specified on a class, function, or field.

- ”Resources”: used to declare a reference to a resource. This annotation acts as a container for multiple resource declarations.

And many others such as “PostConstruct”, “PreDestroy” etc.

How to use annotations

In this section, we will show you how you can use your annotations.

Note:

To be able to work with your annotation (like with this one that we will present in this section’s example) it is required to add Kotlin reflection dependency to your “build.gradle” configuration:

```
dependencies {

    ...

    implementation
        "org.jetbrains.kotlin:kotlin-reflect:1.7.10"
}
```

Or, like in the case of our code examples project (within the main “build.gradle” file):

```
dependencies {

    api "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
    api "org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"
}
```

Without this dependency, your application would crash. With it present, we are ready to write and try our example code. From book code examples open “AnnotationsTutorial.kt”:

```
@Target(AnnotationTarget.CLASS)
annotation class Descriptor(val description: String)
```

We will use this annotation to “describe” items in the online store. Then, we will define some store products:

```
abstract class Product

@Descriptor("vehicle")
class Car : Product()

@Descriptor("office device")
class Computer : Product()

@Descriptor("home appliance")
class Tv : Product()

// No annotation for this one:
class VacuumCleaner : Product()
```

“Product” represents a basic abstraction for the store products. The “Car”, “Computer”, “Tv” and “VacuumCleaner” are the products that will be available in the store for purchase. Each of them is “described” with our annotation except for “VacuumCleaner”. Let’s process products in one shopping cart and handle our annotation:

```
class Processor {
    fun process(products: List<Product>) {

        products.forEach {

            var described = false
            val annotations = it::class.annotations
            annotations.forEach { ann ->
                if (ann is Descriptor) {

                    described = true
                    println(
                        "Processing: ${it::class.simpleName},
                            ${ann.description}")
                }
            }
            if (!described) {

                println("Processing: ${it::class.simpleName}")
            }
        }
    }
}
```

The “Processor” class is responsible for processing items in the shopping cart. It has only one function: to “process”. “process” function iterates through items in the shopping cart and accesses to annotations assigned to each item:

```
val annotations = it::class.annotations
```

Then, the function iterates through all annotations. Since we are only interested in “Descriptor” annotation, if the annotation is present we are obtaining the “description” field value and we are using it further. All other annotations (if present) are ignored.

Let’s try our processor:

```
val products = listOf(

    Tv(),
    Car(),
    Computer(),
    VacuumCleaner()
)

val processor = Processor()
processor.process(products)
```

Executing our program will produce the following output:

```
Processing: Tv, home appliance
Processing: Car, vehicle
Processing: Computer, office device
Processing: VacuumCleaner
```

Any

In this section, we will explain the meaning of “Any” data type. “Any” is a superclass for any class that we may define in our programs, it is the root of the Kotlin class hierarchy.

Besides default “<init>()” (default constructor) “Any” provides mandatory functionalities that all derived classes are inheriting. We will mention each.

“equals” function

Tells us whether some other object is "equal to" the current one. Implementations must fulfill the following requirements:

- Reflexive: for any non-null value “x, x.equals(x)” should return true

- Symmetric: for any non-null values ”x” and “y, x.equals(y)” should return true if and only if y.equals(x) returns true

- Transitive: for any non-null values ”x”, “y”, and ”z”, if “x.equals(y)” returns true and “y.equals(z)” returns true, then “x.equals(z) ” should return true.

- Consistent: for any non-null values ”x” and ”y”, multiple invocations of “x.equals(y)” consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.

- Never equal to “null”: for any ”non-null” value “x, x.equals(null)” should return false.

“hashCode” function

Returns a “hash code” value for the object. The general contract of “hashCode” is:

- The function must consistently return the same integer whenever it is invoked on the same object more than once

- Calling the ”hashCode” function on each of the two objects must produce the same integer result if the two objects are equal according to the “equals” function.

“toString” function

Returns a string representation of the object.

Unit

“Unit” is a special data type. It has only one purpose, it represents the void return type as a class. “Unit” cannot be instantiated since its only constructor is private.

Functions that are returning “Unit” may be represented like this:

```
fun hello(): Unit {

        // ...
}
```

or

```
fun hello() {

        // ...
}
```

As you can see the “Unit” return type declaration is optional.

Nothing

“Nothing” is a special data type (class) in Kotlin. “Nothing” class has no instance and its main purpose is to be used as a value that never exists. It is also used to represent a return type from a function that will never return anything. For example, if a function has the return type of “Nothing”, that means that it never returns and that for example always throws an exception.

Let’s cover this with a simple example. From book code examples open “Nothing.kt”:

```
@Throws(IllegalArgumentException::class)
fun fail(message: String): Nothing {

    throw IllegalArgumentException(message)
}
```

As you probably remember in Kotlin “throw” is an expression and can be used as a part of “Elvis” operator. Here we will nest our function:

```
@Throws(IllegalArgumentException::class)
fun printer(vararg what: String?) =
    what.forEach {
        println(
            it ?: fail("Invalid input, stopping")
        )
    }
```

So, if we pass at least one of the arguments to the “printer” function as a “null”, it will trigger the “fail” function, and “printer” will stop. Let’s try it:

```
try {

    printer("Hello world", null, "Ok")
} catch (e: IllegalArgumentException) {

    println("${e.message}")
}
```

Executing our program will produce the following output:

```
Hello world
Invalid input, stopping
```

Visibility modifiers

In Kotlin classes, objects, interfaces, constructors, functions, properties, and their setter functions can have “visibility modifiers”. “visibility modifiers” are used to set the visibility or access rights. Getter functions always have the same visibility as the property.

In Kotlin, the following “visibility modifiers” are available:

- “private”: visible only inside the current scope
- “protected”: visible for the current scope and all that extend it
- “public”: visible everywhere
- “internal”: visible everywhere inside the current module

If not “visibility modifier” is specified, the default visibility is used which is “public”.

Let’s demonstrate “visibility modifiers” use. From book code examples open “VisibilityModifiers.kt”:

```
open class A {

    fun hello() = println("Hello")

    protected fun world() = println("World")

    private fun helloWorld() = println("Hello world")

    open fun greeting() {

        helloWorld()
    }
}

class B : A() {

    override fun greeting() {

        hello()
        world()
    }
}
```

We have defined two classes: “A” and “B”, where “B” inherits “A”. Both classes define functions with “visibility modifiers”.

“A” class defines the following functions:

- “hello”: it is publicly visible and can be accessed within its scope, from inherited classes, and through the class instance.

- “world”: it is “protected”, which means that can be accessed within its scope and from inherited classes.

- “helloWorld”: it is “private” which means that can be accessed only within its scope.

- “greeting”: is publicly visible, and that means that it has the same rules associated with it as the “hello” function.

“B” class inherits all functions from class “A” and overrides the “greeting” function.

Let’s play with these classes:

```
val a = A()
val b = B()

println("A:")
a.hello()
a.greeting()

println("B:")
b.hello()
b.greeting()
```

If we run this example, the following output will be produced:

```
A:
Hello
Hello world

B:
Hello
```

```
Hello
Worldtors
```

Extensions

Kotlin provides a mechanism for extending classes with new functionalities without inheriting them or using any type of design pattern such as Decorator. This mechanism is called “extensions”.

“Extensions” do not modify parent classes. They are just extending it. Extensions do not insert a new function to a class but create a “callable” inside a class where we are extending.

From book code examples open “Extensions.kt”:

```
class Hello {

    fun world() {

        println("Hello world")
    }
}

fun Hello.everybody() {

    world()
    println("Hello everybody")
}

fun Hello.repeat(count: Int, what: String) {

    for (x in 0 until count) {

        println(what)
    }
}

fun Hello.stranger() {

    println("Hello stranger")
}
```

We have created the “Hello” class that has only one function: “world”. Then, we are extending with three additional functions: “everybody”, “repeat”, and “stranger”. As you can see, the “Hello” class has been extended without inheritance or the “Decorator” pattern.

Let’s try these functions:

```
val hello = Hello()

hello.world()
hello.everybody()
hello.repeat(3, "Woo-hoo")
hello.stranger()
```

Executing our program will produce the following output:

```
Hello world
Hello world
Hello everybody
Woo-hoo
Woo-hoo
Woo-hoo
Hello stranger
```

Let’s see another example. Imagine that you are working on an Android application project. Instead of extending some “BaseActivity” class to give certain functionality to all your activities, you can add functionality to the “Activity” class itself. Let’s say that you need functionality to go back to the previous UI location, you can create an extension for this purpose.

From book code examples open “AndroidExtension.kt”:

```
fun Activity.goBack() {

    // We will pop back stack here...
}
```

All activities (classes that extend the “Activity” class) in your application will have access to the ”goBack” function.

Let’s take a look at the next example. From book code examples open “Extensions2.kt”:

```
fun String.stripDownWithUnderscores(): String {

    return replace(" ", "_")
}
```

Let’s try it:

```
listOf(

    "Hello world",
    "Lorem ipsum"
).forEach {

    val stripped = it.stripDownWithUnderscores()
    println(stripped)
}
```

Which will produce the following output:

```
Hello_world
Lorem_ipsum
```

Extending class properties

In this section, we will show how to extend class properties. From book code examples open “ExetendingProperties.kt”:

```
class ExtendMe {

    val a = 15
}

val ExtendMe.b: Int
    get() = 25
```

We have defined simple class “ExtendMe” with one property: “a”. Then, we have extended it with additional property “b”. Let’s access these properties:

```
val extended = ExtendMe()
```

```
println("A: ${extended.a}")
println("B: ${extended.b}")
```

Executing this small code snippet will produce the following output:

```
A: 15
B: 25
```

Extending objects

In the same way that we have extended the class properties, we can extend objects and companion objects.

We will start with the example of companion object extending. We show you how to do this in the book code example “ExtendingCompanionObject.kt”:

```
class Example {

    companion object {

        const val a = 10

        fun printA() {

            println("A: $a")
        }
    }
}

val Example.Companion.b: Int
    get() = 20

fun Example.Companion.printB() {

    println("B: $b")
}
```

We have defined a simple class called “Example”. “Example” has one constant “a” and one function that prints the value of “a” called “printA”. To introduce the constant “b” and “printB” function without inheritance we are extending the companion object as it is presented in underlined text.

Let’s access these functions:

```
Example.printA()
Example.printB()
```

Executing this code snippet will produce the following output:

```
A: 10
B: 20
```

The next that we are going to demonstrate is how to extend objects as class members. From book code examples open “ExtendingObjects.kt”:

```
interface Drawing {

    fun draw()
}

class Objects {

    val circle = object : Drawing {

        override fun draw() {

            println("Circle")
        }
    }
}

val Objects.square: Drawing
    get() = object : Drawing {

        override fun draw() {

            println("Square")
        }
    }
```

We have defined a simple interface “Drawing” which will be implemented by object class members. Class “Objects” defines one object called “circle” which implements this interface. Then, we are extending the “Objects” class with another object member called “square”. As you can see “square” implements the same interface as “circle”.

Let’s play with this:

```
val o = Objects()
with(o) {

    circle.draw()
    square.draw()
}
```

Executing the example will produce the following output:

```
Circle
Square
```

Note:

In this example, we have used the “width” inline function which calls the specified function block with the given receiver as its receiver and returns its result. We will present more examples of “with” in upcoming sections of the book.

Extension function literals

As you probably remember, In Kotlin there are two commonly used types of function literals:

- Lambdas
- Anonymous functions.

Then you can assume that we can create extensions for the function literals. This is true. We will illustrate this with a simple example same way as we did with other extension types. From book code examples open “ExtendingFunctionLiterals.kt”:

```
class Printer() {

    fun print(what: String) {

        println(what)
    }
}

val printQuoted: Printer.(what: String) -> Unit = {

    what ->
    print("\"$what\"")
}
```

We have defined a simple class called “Printer”. “Printer” class has only one function: “print”. “print” function just prints out the value of the parameter. However, we want to create a new function that will print the quoted value of the parameter. For this purpose, we are extending the “Printer” class with the lambda property which accepts a string as an argument and then it prints it in quotes. Inside the block of the lambda, we can access the “print” member function. Also, we have stated that the return type of the lambda is “Unit” which means that there will be no return value.

Let’s try our extension:

```
val printer = Printer()
printer.print("Hello world")
printer.printQuoted("Lorem ipsum")
```

Executing our program will produce the following output:

```
Hello world
"Lorem ipsum"
```

Scope functions

Scope functions are functions whose purpose is to execute a block of code within the context of an object. Calling a scope function on the object with a lambda expression forms a temporary scope. Inside the temporary scope, we can access the object without its name. Scope functions differ between each other by the way they refer to the context object and by the return value.

Scope functions in Kotlin are the following: ”let”, ”run”, “with”, ”apply”, and ”also”.

This and It

Inside the lambda of a scope function, we can access to the context object by a short reference instead of its actual name. There are two ways to access the context object:

- As a lambda receiver by using “this”
- As a lambda argument by using “it”.

In both cases same capabilities are available.

Using “this”

Scope functions ”run”, ”with”, and “apply” access the context object as a lambda receiver by using the keyword “this”. In lambdas of these scope functions, the object is available the same way as it would be in the ordinary class functions. Generally, you can leave out the “this” keyword when accessing the members of the receiver object. By doing so you will simplify your code.

Using “it”

Scope functions ”let” and “also” have their context object as a lambda argument. If we do not specify the argument name, the object is accessed by the implicit default name.

Scope functions return value

One of the differences between scope functions is by the result they are returning. “apply” and “also” scope functions are returning the context object. Thanks to this, these functions can be included in the call chains. “apply” and “also” scope functions also can be used as return results of functions that are returning the context object. “let”, “run”, and “with” scope functions are returning the lambda result. These functions can be used when assigning the result to a variable or a constant, for chaining of operations on the result, etc. It is also possible to ignore the return value and use the function to create a temporary scope for variables.

Based on this you can make the decision depending on what you need to do in your code block.

“Let” scope function

For the “let” scope function context object is available as an argument. The default argument name is “it”. The “let” scope function return value is the lambda result. Usually “let” can be used to trigger one or multiple functions as a result of call chains.

Let’s demonstrate how to use the “let” scope function with a simple example. From book code examples open “Let.kt”:

```
fun filterShort(text: String): String? {

    if (text.length >= 5) {

        return text
    }
    return null
}

listOf(
    "Hello",
    "world",
    "i",
    "am",
    "here"
).forEach { word ->

    val filtered = filterShort(word)
    filtered?.let {

        println(it)
    }
}
```

We have defined a simple function called “filterShort”. The function returns null or the value of the passed argument depending on its length. Then, we have created a list of the short word through which we are iterating. Each element is passed to the ‘filterShort” function. Thanks to the “let” function we are printing the “filtered” value only if it is not null.

If we execute this simple program the following result will be produced:

```
Hello
world
```

“Width” scope function

“With” scope function represents a non-extension function. The context object of the scope function is passed as an argument inside the lambda. In this case, it is available as a receiver via “this”. The return value of the scope function is the lambda result.

Let's see how we can use the “with” scope function to access functions of some instances. From book code examples open “With.kt”:

```
class Human {
    fun walk() {
        println("Walking");
    }

    fun talk() {
        println("Talking");
    }

    fun jump() {
        println("Jumping");
    }

    fun swim() {
        println("Swimming");
    }
}
```

“Human” is a simple class with several public functions exposed. We will create an instance of “Human“ and access to these functions using the “with” scope function:

```
val human = Human()
with(human) {

    walk()
    walk()
    walk()
    jump()
    swim()
    jump()
    walk()
    walk()
    talk()
}
```

Executing this program will produce the following output:

```
Walking
Walking
Walking
Jumping
Swimming
Jumping
Walking
Walking
Talking
```

"Run" scope function

With the "run" scope function the context object is available as a receiver. In this case, it is available as a receiver via "this". The return value of the "run" scope function is the lambda result. "run" scope function behaves similar to the "with" scope function. The main difference is that it invokes as "let" scope function, as an extension function of the context object.

Let's have a look at an example that illustrates the use of the "run" scope function. From book code examples open "Run.kt":

```
class User {

    var firstName = ""
    var lastName = ""
    var address = ""
    var email = ""
}

val user = User()
user.run {

    firstName = "John"
    lastName = "Smith"
    address = "5th Avenue"
    email = "john.smith@example.com"
}
```

We have defined a simple class called “User” with a couple of publicly visible variables. Then, we have accessed them using the “run” scope function and assigned them values. As you can see it is quite simple to use the “run” scope function.

There is one more application of the “run” function. You can use the “run” scope function as a non-extension function. The non-extension “run“ scope function makes it possible that you execute a block of several statements where an expression is required:

```
val example = run {

    val param1 = "Hello world"
    val param2 = "Lorem ipsum"
    val param3 = "Something else"

    "$param1, $param2, $param3"
}
```

Let’s print the value of the “example”:

```
println(example)
```

Which will produce the following output:

```
Hello world, Lorem ipsum, Something else
```

“Apply” scope function

With the “apply” scope function context object is available as a receiver via “this”. The return value of the “apply” scope function is the object itself. The “apply” scope function is commonly used for code blocks that don't return a value.

Let’s have a look at an example of the use of the “apply” scope function. From book code examples open “Apply.kt”:

```
class User {

    var firstName = ""
    var lastName = ""
    var address = ""
    var email = ""

    override fun toString(): String {

        return "User(firstName='$firstName',
                    lastName='$lastName', address='$address',
                    email='$email')"
    }
}

val user = User().apply {

    firstName = "John"
    lastName = "Smith"
    address = "5th Avenue"
    email = "john.smith@example.com"
}
```

We have extended the “User” class from the previous section to print human-readable representations of its instance. Then, we have assigned values to each of the “User” class properties by using the “apply” scope function. “user” constant has a value of “User” instance with values to its properties assigned. Let’s check them out:

```
println(user)
```

Which will produce the following output:

```
User(firstName='John', lastName='Smith',
    address='5th Avenue', email='john.smith@example.com')
```

“Also” scope function

With the “also” scope function the context object is available as an argument via “it”. The return value of the “also” scope function is the object itself.

Let’s illustrate the typical use case of the “also” scope function. From book code examples open “Also.kt”:

```
class Hello {

    fun execute() = println("Hello!")
}

Hello()
    .also {

        println("Hey!")
    }.execute()
```

As you can see using the “also” scope function can be observed like performing certain operations before the main one. Or simply: “...and also do the following with the instance”.

In this example, we will print “Hey!” before the “execute” function is triggered. Executing this code snippet will produce the following output:

```
Hey!
Hello!
```

“takeIf” and “takeUnless”

The “takeIf” and “takeUnless” functions allow you to incorporate checks of the object state in call chains. “takeIf” returns “this” object if it matches the predicate when it is called on an object with a predicate provided. Otherwise, the “takeIf” function returns null. On the other hand, the “takeUnless” function returns the object if it doesn't match the predicate and null if it does match. The object is available as a lambda argument via “it”.

Let’s see simple example of use for these functions. From book code examples open “TakeIfUnless.kt”:

```
val numbers = listOf(-3, -2, -1, 0, 1, 2, 3)

numbers.forEach {
    it.takeUnless { it == 0 }
        ?.let { nZero ->

            nZero.takeIf { nZero > 0 }
                ?.let { pos ->

                    println("Positive, non zero number: $pos")
                }
        }
}
```

What we are practically doing in this example is filtering of all positive non-zero numbers. Executing this example will produce the following output:

```
Positive, non zero number: 1
Positive, non zero number: 2
Positive, non zero number: 3
```

Singleton pattern in Kotlin

Singleton pattern is one of the most frequently used design patterns in modern software development. Singleton pattern depends on a single class that is responsible to create a class instance (the object) while it makes sure that only one instance exists (is created).

In Kotlin, we can use objects to create singletons. From book code examples open “Singleton.kt”:

```
object Single {

    var value = 0

    fun print() {
        println("Value: $value, Hash: ${this.hashCode()}")
    }
}
```

As you can see we have a defined object called “Single”. The object contains one variable and one function. Let’s assign it to some constants and access these object members:

```
val s1 = Single
val s2 = Single
val s3 = Single

s1.print()
s2.print()
s3.print()

s1.value = 100

s1.print()
s2.print()
s3.print()
```

Executing this will produce the following output:

```
Value: 0, Hash: 1872034366
Value: 0, Hash: 1872034366
Value: 0, Hash: 1872034366
Value: 100, Hash: 1872034366
Value: 100, Hash: 1872034366
Value: 100, Hash: 1872034366
```

You will notice that each constant ("s1", "s2", and "s3") has the same hash code which means that it points to the same reference. Objects occupy memory as a single instance which makes them a perfect candidate for singleton pattern implementation.

Lazy initialization

Lazy initialization represents a delaying of instance creation, the calculation of a value, or some other expensive process until the first time it is needed (access to the variable for example).

Kotlin supports lazy initialization. We will illustrate this with two simple examples. From book code examples open our first example of "lazy" initialization "DbManager.kt":

```
object DbManager {
    private const val dbName = "cars"
    private const val dbVersion = 1

    val database: SQLiteDatabase by lazy {

        DbHelper(dbName, dbVersion).getWritableDatabase()
    }
}
```

The “database” field will not be instantiated until the first time it is accessed. It can be direct access to the field or by any function that uses the field. Instantiation block will be executed only the first time when we access the “database” field.

Let’s have a look at another simple example. From book code examples open “Lazy.kt”:

```
class Example {

    val hello: String by lazy {

        println("I am initializing this lazy value")
        "Hello"
    }
}
```

Again, we have to define a simple class with the constant which will be lazily initialized. Inside its initialization block, we will print simple “log” information and return the value that will be assigned to the constant.

Note:

The evaluation of lazy properties **is synchronized by default**. The value is produced only in one thread, and all threads will see the same value.

Let’s try our class:

```
val example = Example()
println(example.hello)
```

Executing this code snippet will produce the following output:

```
I am initializing this lazy value
Hello
```

Properties with late initialization

Properties can be late-initialized. The main difference when it comes to comparing with lazy initialization is that late-initialized properties are variables. Let’s take a look at the example and see how properties can be late-initialized. From book code examples open “LateInit.kt”:

```
class X {

    init {

        println("I am late initialized")
    }

    fun doSomething() = print("Hello")
}

class Y {

    lateinit var x: X

    fun doSomething() {

        x = X()
        x.doSomething()
        print(" World!")
    }
}
```

Since it is expected that properties are initialized in the constructor, sometimes that is not what we want. For that particular situation, we mark our property with the “lateinit” keyword. The “lateinit” may be only used with var properties that are defined in the class body with no custom getter and setter. The type of property must not be non-null or primitive.

Let’s check how this works:

```
val y = Y()
y.doSomething()
```

Executing this small code snippet will produce the following output:

```
I am late initialized
Hello World!
```

Delegating behavior

In Kotlin we can delegate class behavior to the other classes. This functionality comes supported in Kotlin out of the box. The delegation pattern makes our code more flexible and it is proven when it comes to finding the perfect alternative to the inheritance.

Let's take a look at the simple example of behavior delegation. From book code examples open "Delegate.kt":

```
interface Flying {

    fun fly()
}
```

"Flying" will be our very basic definition of behavior with two implementations:

```
class Plane : Flying {

    override fun fly() {
        println("PLANE")
    }
}

class Zeppelin : Flying {

    override fun fly() {
        println("ZEPPELIN")
    }
}
```

Next that we do is definition of a class that **can implement an interface by delegating all of its public members to a specified object**:

```
class Traveling(fly: Flying) : Flying by fly
```

Let’s play with this:

```
val plane = Plane()
val zeppelin = Zeppelin()

val travelByPlane = Traveling(plane)
val travelByZeppelin = Traveling(zeppelin)

travelByPlane.fly()
travelByZeppelin.fly()
```

Executing our program will produce the following output:

```
PLANE
ZEPPELIN
```

Delegating properties

In Kotlin we can delegate class properties. There are some cases when we need them, for example when we need the “lazy” initialization.

The mechanism that we are using to achieve this is very similar to one from the previous section. From book code examples open “DelegateProperties.kt”:

```
class Delegator : ReadWriteProperty<Any, Int> {

    var value = 0

    override fun getValue(
      thisRef: Any, property: Kproperty<*>
    ): Int {

        println("Value GET: ${property.name} -> $value")
        return value
    }

    override fun setValue(
      thisRef: Any, property: KProperty<*>, value: Int
    ) {

        println("Value SET: ${property.name} -> $value")
        this.value = value
    }
}

class Data {

    var value: Int by Delegator()
}
```

The “Delegator” class will be responsible for handling read and write operations for the integer properties. “Delegator” implements the “ReadWriteProperty” interface required for this functionality. We will soon mention the most commonly used delegation interfaces. Finally, the “Data” class contains a “value” variable. Read and write operations for the variable will be delegated to the instance of the “Delegator” class.

Delegation syntax is the following:

val/var <property name>: <Type> by <expression>

The expression after **by** is the delegate. The get() / set() corresponding to the property will be delegated to its getValue() and setValue() functions.

These are the main delegate types:

Delegates, object Delegates

Standard property delegates:

- **ObservableProperty**,

```
abstract class ObservableProperty<T> :
    ReadWriteProperty<Any?, T>
```

Implements the core logic of a property delegate for a read/write property that calls callback functions when it is changed.

- **ReadOnlyProperty**,

```
interface ReadOnlyProperty<in R, out T>
```

The base interface that can be used for implementing property delegates of read-only properties.

- **ReadWriteProperty**,

```
interface ReadWriteProperty<in R, T>
```

The base interface that can be used for implementing property delegates of read-write properties.

Let’s try now classes from our last example:

```
val x = Data()
val y = Data()
val z = Data()

x.value = 100
y.value = 200
z.value = 300

println("x's val: ${x.value}")
println("y's val: ${y.value}")
println("z's val: ${z.value}")
```

Executing example code will produce the following output:

```
Value SET: value -> 100
Value SET: value -> 200
Value SET: value -> 300
Value GET: value -> 100
x's val: 100
Value GET: value -> 200
y's val: 200
Value GET: value -> 300
z's val: 300
```

Property delegation requirements

To be able to delegate objects we must fulfill the following requirements:

- For read-only property delegate has to provide a “getValue” function that takes parameters: receiver (the same or a super-type of the property owner) and metadata (“KProperty<*>” or its super-type).

- For mutable property delegate has to provide one more function: “setValue” that takes parameters: receiver (same as above), metadata (same as above), and new value to be set.

These functions can be provided in two ways:

- As member functions of delegated class
- As extension functions.

Observable

The “Observable” delegate is one of the most commonly used delegates in everyday development. The ”observable” function of the delegate returns a property delegate for a read and write property which calls a specified callback function when changed. From book code examples open “Observable.kt”:

```
class Example {

  val default = "No value"

  var value: String by Delegates.observable(default) {
        property, oldValue, newValue ->

      println(
            "${property.name}: old='$oldValue',
            new='$newValue'"
      )
  }
}
```

Let’s try this delegate and see what will be the final output of the program:

```
val example = Example()
example.value = "Hello world!"
example.value = "Lorem ipsum..."
example.value = "And so on..."
```

Executing the program will produce this:

```
value: old='No value', new='Hello world!'
value: old='Hello world!', new='Lorem ipsum...'
value: old='Lorem ipsum...', new='And so on...'
```

Builders

In this section, we will show you the two most frequently used building mechanisms. They are used to build strings and to build maps. We will cover them through sections:

- String builder
- and map builders.

String builder

String builder is used for performing multiple string manipulation operations. To better understand possibilities of the string builder open “StringBuilder.kt” from book code examples:

```
val builder = StringBuilder()
    .append("Hello world!")
    .append("\n")
    .append("Numbers: ")

val numbers = StringBuilder()
for (number in 1..3) {

    numbers.append(number)
        .append(" ")
}

val text = builder.append(numbers)
    .append("\n")
    .append("Lore ipsum...")
    .toString()

println(text)
```

We have created the “StringBuilder” class instance on which we are chaining “append” function calls. Since the “append” function returns the current instance (“this”) chaining is possible. We can append any base data type such as string, integer, character, and so on. Also, we can append another builder. We did that with the “numbers” string builder that concatenated integer numbers. Finally, we have converted the builder into the string by calling the “toString” function. Executing this code example will produce the following output:

```
Hello world!
Numbers: 1 2 3
Lore ipsum...
```

Map builders

In this section, we will show you five functions that can be used to instantiate maps.

- “mapOf”, creates an immutable map of the key-value pairs that we provide as the parameters
- “mutableMapOf”, creates a mutable map of the key-value pairs we provide as parameters
- “sortedMapOf”, creates a map that is ordered by its keys
- “hashMapOf”, creates a hash table based implementation of the map
- “linkedMapOf”, creates a “LinkedHashMap” variant of a hash table implementation.

Let’s try each of these. From book code examples open “MapBuilders.kt”:

```
val map1 = mapOf(
    "hello" to "world", "lorem" to "ipsum", "abc" to "def"
)

val map2 = mutableMapOf(
    "hello" to "world", "lorem" to "ipsum"
)
map2["abc"] = "def"

val map3 = sortedMapOf(
    "hello" to "world", "lorem" to "ipsum", "abc" to "def"
)
val map4 = hashMapOf(
    "hello" to "world", "lorem" to "ipsum", "abc" to "def"
)
```

```
val map5 = linkedMapOf(
    "hello" to "world", "lorem" to "ipsum", "abc" to "def"
)
```

As you can see we have created maps by using every of these five functions. Every function creates a different map. Let’s see the content of each map:

```
println("mapOf=$map1")
println("mutableMapOf=$map2")
println("sortedMapOf=$map3")
println("hashMapOf=$map4")
println("linkedMapOf=$map5")
```

Executing this code snippet will produce the following output:

```
mapOf={hello=world, lorem=ipsum, abc=def}
mutableMapOf={hello=world, lorem=ipsum, abc=def}
sortedMapOf={abc=def, hello=world, lorem=ipsum}
hashMapOf={lorem=ipsum, hello=world, abc=def}
linkedMapOf={hello=world, lorem=ipsum, abc=def}
```

Destructuring

In Kotlin it is possible to “destructure” class into several variables. For example:

```
val (brand, model) = car
```

This can be very convenient if we want to return multiple results from the function. Let’s say we define a data class holding values representing function result, we may use it like this:

```
val (a, b, c) = myFunction()
```

Also, we can use the same approach for loops:

```
for ((a, b) in collection)
```

or

```
for((key, value) in map)
```

As you can see ”destructuring” is one nice little functionality given to you by Kotlin.

Mapped properties

In Kotlin we might want to store our properties in a map. This can be easily achieved. From the book code examples open “MappedProperties.kt”:

```
class Employee(map: Map<String, Any?>) {

   val firstName: String by map
   val lastName: String by map
   val yearOfBirth: Int by map
}

val e = Employee(

   mapOf(

      "firstName" to "John",
      "lastName" to "Smith",
      "yearOfBirth" to 1985
   )
)
```

Let’s check if values have been assigned properly:

```
println(

   "Employee: ${e.firstName}, ${e.lastName},
   ${e.yearOfBirth}"
)
```

Executing this verification code snippet will confirm that everything has been assigned as we have expected:

```
Employee: John, Smith, 1985
```

Concurrency

In this section, we will explain what concurrency is and how concurrency can be performed in Kotlin. The definition of concurrency is the ability of different parts of our program to be executed unordered or at the same time simultaneously or in a partial order, without affecting the outcome of the execution. Thanks to this our programs can significantly improve speed on multi-processor and multi-core machines.

We will demonstrate two approaches to concurrency:

- by using “threads”, the classical approach
- and by using Kotlin “coroutines”.

We will explain how to use each of these two, what are their strengths and what their weaknesses are.

Threads

A “thread” represents a thread of execution in our programs. The JVM allows that our programs run multiple threads at the same time. All threads of our program have a priority. This means that the threads with high priority are executed in preference to the lower priority ones. Another thing that is possible is to mark the thread as a daemon.

When the JVM starts a single non-daemon thread is started. It calls the main function of our program. Then, the JVM continues to execute our program threads until one of the conditions happen:

- The “exit” function of the “Runtime” class has been called (with the “security manager” permitted the exit operation)
- All threads that are not daemon threads have died.

We will present you two ways to create and run threads:

- Declaring a class as a subclass of the “Thread” and starting it
- Declaring a class that implements the “Runnable” interface which then implements the “run” function and using it as a parameter to the “Thread” class constructor.

From book code examples open “Threads.kt”:

```
class Counter(private val count: Int) : Thread() {
    override fun run() {

        for (x in 0..count) {

            println("Count no. $x")
            // Sleep for one second:
            sleep(1000)
        }
    }
}
```

Class “counter” represents the first approach in creating threads. It extends the “Thread” superclass and it implements the “run” function which will be executed in the background once the “start” function is triggered. What our implementation does is simple counting with pauses of one second in between print lines.

Let’s see how this works:

```
val counter = Counter(10)
counter.start()

// We will wait until counter finishes:
while (counter.isAlive) {

    Thread.yield()
}
```

Once the “start” function is triggered we will count and append lines to the output. For all output lines to print out it will take around ten seconds:

```
Count no. 0
Count no. 1
Count no. 2
Count no. 3
Count no. 4
Count no. 5
Count no. 6
Count no. 7
Count no. 8
Count no. 9
Count no. 10
```

We can rewrite this same example to the second approach:

```
class Counter2(private val count: Int) : Runnable {

    override fun run() {

        for (x in 0..count) {

            println("Count no. $x (2)")
            // Sleep for one second:
            Thread.sleep(1000)
        }
    }
}
```

We have defined the class “Counter2” which inherits the “Runnable” interface. Everything else is straightforward. The “run” function performs the same operation as in our previous example. Let’s try it:

```
val counter2 = Counter2(10)
val thread = Thread(counter2)
thread.start()

// We will wait until counter finishes:
while (thread.isAlive) {

    Thread.yield()
}
```

As you can see we have created a new object of the “Thread” class and passed a new counter as the constructor’s parameter. Then, instead of calling the “start” function on the counter object, we have called on the thread directly. Executing this code snippet will give us the same behavior as in the previous example:

```
Count no. 0 (2)
Count no. 1 (2)
Count no. 2 (2)
Count no. 3 (2)
Count no. 4 (2)
Count no. 5 (2)
Count no. 6 (2)
Count no. 7 (2)
Count no. 8 (2)
Count no. 9 (2)
Count no. 10 (2)
```

Thread execution

Creating a new thread is an expensive operation. To create a thread operating system allocates resources that are needed for the thread. Let’s say that you pile up a huge amount of threads that are running at the same time. This could choke your machine. For this not to happen we execute threads in a controlled manner. In practice “thread pools” are used to utilize system resources efficiently and gain maximal performance.

The thread pool does not create new threads when a new task arrives. A thread pool keeps several idle threads. Once the new task has arrived, these threads are ready to execute. Once a thread finishes, the thread pool thread does not die. It goes back to the idle state in the pool and it waits for the next task to come.

Thanks to limits that we can define, we can limit the number of concurrent threads in the pool that are being executed at the same time. Thanks to this we can prevent overload and choking of the system. In the situation when all threads are busy, the latest arrived tasks are placed in a queue. Once the thread becomes available the task is taken from the queue and executed.

There are several types of thread pools commonly used:

- Cached thread pool: it keeps a certain number of alive threads, it creates new ones if required
- Fixed thread pool: introduces a limit for the maximum of concurrent threads while newly arrived tasks are waiting in a queue
- Single-threaded pool: has only one thread that executes one task at a time
- Fork/Join pool: a special thread pool that uses the “Fork/Join” framework.

We will illustrate the use of thread pools in the example of a single-threaded pool. From book code examples open “Executors.kt”:

```
val executor = Executors.newSingleThreadExecutor()
```

As you can see getting the instance of the single-threaded pool is straightforward. Besides the “newSingleThreadExecutor” function there are others as well required to create other types of thread pools:

- “newCachedThreadPool”: creates a cached thread pool
- “newFixedThreadPool”: creates a thread pool that reuses a fixed number of threads.

Let’s schedule some work for our thread pool:

```
for (x in 0..10) {

    executor.execute {

        println("Count no. $x")
        // Sleep for one second:
        Thread.sleep(1000)

        if (x == 10) {

            // We terminate the executor:
            executor.shutdown()
        }
    }
}
```

And let’s wait for it to finish:

```
// We wait for the executor to be terminated
// and to finish all tasks:
while (!executor.isShutdown) {

    Thread.yield()
}
```

Executing our program will count to ten and produce the following output (writing the output line per second):

```
Count no. 0
Count no. 1
Count no. 2
Count no. 3
Count no. 4
Count no. 5
Count no. 6
Count no. 7
Count no. 8
Count no. 9
Count no. 10
```

Coroutines

It is time to face the concurrency in a more Kotlin idiomatic way. We will demonstrate the use of Kotlin coroutines. Coroutines represent instances of computations that can be paused. Kotlin coroutines are similar to threads. A coroutine executes a block of code that runs concurrently with the rest of the program. On the other side, a coroutine is not bound to any particular thread. It can pause the execution in one thread and resume it some other.

To be able to use Kotlin coroutines it is required to add coroutines dependency into your “build.gradle” configuration:

```
implementation
    "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.8"
```

or

```
api "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.8"
```

Let’s have a look at a simple coroutine example. From book code examples open “Coroutines.kt”:

```
// Coroutine scope (class: 'CoroutineScope'):
runBlocking {

    // Launch a new coroutine and continue:
    launch {

        for (x in 0..10) {

            // Non-blocking delay for one second
            delay(1000L)
            println("Count no. $x")
        }
    }

    // Main coroutine continues,
    // a previous one is delayed:
    println("Hello Coroutines!")
}
```

There are several important points to explain in this code snippet:

- “launch” represents a coroutine builder. Coroutine builder launches a new coroutine that runs concurrently with the rest of the code. The rest of the code continues to work independently.

- “delay” is a pausing function. It pauses the coroutine for a specific time in milliseconds. This does not block the underlying thread but it allows the other coroutines to run.

- “runBlocking” is another coroutine builder that ties the main program function and the coroutine code block. “runBlocking” means that the thread that is currently running is blocked, until all the coroutines inside “runBlocking” complete their work.

Let’s run our program:

```
Hello Coroutines!
Count no. 0
Count no. 1
Count no. 2
Count no. 3
Count no. 4
Count no. 5
Count no. 6
Count no. 7
Count no. 8
Count no. 9
Count no. 10
```

Coroutines follow a principle of “structured concurrency”. The “structured concurrency” principle means that new coroutines can be only launched in a specific coroutine scope (“CoroutineScope” is the class). The scope determines the lifetime of the coroutine. The “structured concurrency” makes sure that there is no loss or leak. Outer scopes cannot complete until all of the children are complete. The “structured concurrency” also makes ensures that errors that may occur are properly reported and that they are not lost.

Coroutine scope building

Scope builder is a powerful utility used in any suspending function to execute concurrent operations. We will illustrate this with a simple example. From book code examples open “CoroutinesScopeBuilder.kt”:

```
fun main() = runBlocking {

   fire()
   println("Completed")
}

// 'suspend' means that coroutine can be suspended
// for the later execution
suspend fun fire() = coroutineScope {

   for (x in 0..3) {

       launch {

           for (y in 0..5) {

               delay(1000L)
               println(
                   "Job $x, count no. $y at
                   ${System.currentTimeMillis()}"
               )
           }
       }
   }

   println("Started")
}
```

In this example, each launch block executes concurrently. A “coroutineScope” in our “fire” function completes once each “launch” block has finished. Let’s run it:

```
Started
```

```
Job 0, count no. 0 at: 1621087584784
Job 1, count no. 0 at: 1621087584787
Job 2, count no. 0 at: 1621087584787
Job 3, count no. 0 at: 1621087584787
Job 0, count no. 1 at: 1621087585784
Job 1, count no. 1 at: 1621087585787
Job 2, count no. 1 at: 1621087585787
Job 3, count no. 1 at: 1621087585787
Job 0, count no. 2 at: 1621087586788
Job 1, count no. 2 at: 1621087586788
Job 2, count no. 2 at: 1621087586788
Job 3, count no. 2 at: 1621087586789
Job 0, count no. 3 at: 1621087587792
Job 1, count no. 3 at: 1621087587792
Job 2, count no. 3 at: 1621087587793
Job 3, count no. 3 at: 1621087587793
Job 0, count no. 4 at: 1621087588793
Job 1, count no. 4 at: 1621087588793
Job 2, count no. 4 at: 1621087588793
Job 3, count no. 4 at: 1621087588793
Job 0, count no. 5 at: 1621087589795
Job 1, count no. 5 at: 1621087589796
Job 2, count no. 5 at: 1621087589796
Job 3, count no. 5 at: 1621087589796
Completed
```

Coroutine job

In this section dedicated to coroutines, we will show you how coroutines “Job” is used. The coroutine builder launch returns an instance of the “Job” class. “Job” represents an access mechanism to the launched coroutine. Thanks to this it can be used to explicitly wait for execution completion.

Let’s demonstrate this. From book code examples open “CoroutinesJob.kt”:

```
fun main() = runBlocking {

    // We are launching a new coroutine
    // and keep a reference to coroutine’s “Job”:
    val job = launch {

        for (x in 0..5) {
            delay(1000L)
            println("Count: $x")
        }
    }

    println("Start")
    // Wait for coroutine to finish:
    job.join()
    println("End")
}
```

As you can see the “join” function suspends the coroutine until this job is complete. This invocation resumes normally when the job is complete for any reason. The Job of the invoking coroutine is still active. If the Job was still in a new state, this function also starts the corresponding coroutine.

The job becomes complete only when all of its children are complete! This suspending function is cancellable. It always checks for a cancellation of the invoking coroutine's Job. If the Job of the invoking coroutine is canceled or completed when this suspending function is invoked or while it is suspended, this function throws “CancellationException”.

It means that a parent coroutine invoking join on a child coroutine that was started using:

```
launch(coroutineContext) {

    // ...
}
```

builder throws “CancellationException” if the child had crashed unless a non-standard “CoroutineExceptionHandler” is installed in the context.

This function can be used in select invocation with the “onJoin” clause. Use “isCompleted” to check for the completion of this job without waiting. Also, there is the “cancelAndJoin” function that combines an invocation of cancel and the join.

Let’s run our program:

```
Start
Count: 0
```

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
End
```

Summary

In sections and examples that we have provided in this book, we were focused on the basic usage of the Kotlin programming language. The main goal was to demonstrate how to use it as a tool for a modern developer and to present its strengths and most valuable functionalities. Most of the sections were not focused on the deep theory behind each explained Kotlin feature but short, clear, and concise code demonstration. At the end of the day, what the developer needs more than a proper code example?

Miloš Vasić,
Summer 2021.