Compile to Bytecode and Run

We'll cover the following

- Running on the command line
- Running in IDEs
- Experiment with the REPL
- Run as a script

Let's first create a small Hello World program in Kotlin. Using your favorite text editor create a file named `Hello.kt`, like so:

```
fun main() = println("Hello World")
```

Don't worry about the syntax in the code at this time; let's keep our focus on getting the code to run. We'll discuss the language syntax and semantics in following chapters. You may specify the parameter for the `main()` function if you like, but starting with `Kotlin 1.3`, it's optional. If you're using a version prior to `1.3`, then you'll need to add the parameter, like so:

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

Running on the command line

To compile and run the code from the command line, first execute the following command:

```
kotlinc-jvm Hello.kt -d Hello.jar
```

This command will compile the code in the file `Hello.kt` into Java bytecode and place that into the Hello.jar file.

Once the jar file is created, run the program using the java tool, like so:

Once the jar file is created, run the program using the java tool, like so:

```
java -classpath Hello.jar HelloKt
```

Since the file `Hello.kt` contains only the main function and not a class, the Kotlin compiler, `kotlinc-jvm`, automatically creates a class named after the file name, without the `.kt` extension, but adds a `Kt` suffix. Here's the output of running the code:

```
Hello World
```

Instead of specifying the classpath command-line option, you may also use the jar option to run the code. That's because, upon finding the `main()` function, the Kotlin compiler decided to add the Main-Class manifest attribute to the jar file. Go ahead and try out the following command on the `Hello.jar` file:

```
java -jar Hello.jar
```

The output of this command will be the same as the output produced when the classpath option was used instead of the jar option.

In this example we didn't use anything from the Kotlin standard library. However, any nontrivial program will make use of classes and functions from the Kotlin standard library and, in that case, the above execution using the java tool will fail with a `java.lang.NoClassDefFoundError` exception. To avoid this, include the `kotlin-stdlib.jar` file to the classpath, like so:

```
kotlinc-jvm Hello.kt -d Hello.jar

java -classpath Hello.jar:$KOTLIN_PATH/lib/kotlin-stdlib.jar HelloKt
```

The environment variable `$KOTLIN_PATH` on Unix-like systems, or `%KOTLIN_PATH%` on Windows, refers to the directory where Kotlin is installed. On Windows, instead of `:`, use `;` to separate the paths in the classpath.

Instead of using the java tool, you may also use the kotlin tool. In this case, you don't have to refer to the `kotlin-stdlib.jar`. Let's run the code using kotlin. Here are the steps, but you may skip the first step; it's the same compilation command as before:

```
kotlinc-jvm Hello.kt -d Hello.jar
```

```
kotlinc-jvm Hello.kt -d Hello.jar

kotlin -classpath Hello.jar HelloKt
```

The output of the code will be the same whether we run the code using the java tool or kotlin tool.

Use the java tool if you're predominantly programming in Java and mixing Kotlin on your project. Otherwise, use the kotlin tool as that needs fewer configuration options.

Running in IDEs

It should be no surprise that IntelliJ IDEA from JetBrains, which is also the company behind Kotlin, has excellent support for programming with the language. Some developers think they need that IDE to use Kotlin, but that's not true. Kotlin doesn't require you to use a particular IDE, or any IDE for that matter.

Kotlin is bundled with the newer versions of IntelliJ IDEA and also with the free and open source IntelliJ IDEA Community edition. To use IntelliJ IDEA for development with Kotlin, start by creating a Kotlin project. Once you create a project, you can quickly create Kotlin files and execute with a few mouse clicks or keyboard shortcuts. The short tutorial on the official website for the language will give you a quick jump start if you get stuck with any steps.

If you're an Eclipse aficionado, then you may use Eclipse Neon or later to program with Kotlin. The official Kotlin language website has a tutorial for Eclipse, as well, to help get you started with Kotlin on Eclipse.

NetBeans fans can benefit from a NetBeans Kotlin Plugin to program Kotlin applications.

Check to make sure you're using the right version of IDE and the correct version of Kotlin supported in that version.

Experiment with the REPL

Several languages provide a read-evaluate-print loop (REPL) command-line shell to run small snippets of code. I like to call them micro-prototyping tools. When you're in the middle of coding or making your current automated test pass, instead of wondering what a particular function does, you can quickly take it for a ride in the REPL. Once you verify that a small piece of code is what you're looking for, you can use the best tool humans have invented—copy-and-paste—to bring that over from

the REPL to your editor or IDE. The interactive tool is also very useful to show a colleague how a small piece of code works, without having to create a project in an IDE, for instance.

The Kotlin compiler that we used previously, kotlinc-jvm, turns into a REPL shell when run without any options or file names. Let's run an interactive session. From the command line, type the command kotlinc-jvm and the REPL will respond with a prompt for you to key in some code. Type in some code, like in the following interactive session, and observe the response:

```
kotlinc-jvm

Welcome to Kotlin ...
Type :help for help, :quit for quit
>>> 7 + 5
res0: kotlin.Int = 12
>>> val list = listOf(1, 2, 3)
>>> list.map { it * 2 }
res2: kotlin.collections.List<kotlin.Int> = [2, 4, 6]
>>>
```

As soon as you key in a snippet of code and hit the enter key, the REPL will evaluate that piece of code, display the response, and prompt you for the next snippet. When you're done, hit `ctrl+d` (`ctrl+c` on Windows) or key in `:quit` to terminate the REPL session.

From within the REPL, you may also load existing files to execute code in them. For example, let's load the `Hello.kt` file we created earlier and run it within the REPL, without going through an explicit compilation step.

```
kotlinc-jvm

Welcome to Kotlin ...
Type :help for help, :quit for quit
>>> :load Hello.kt
>>> main()
Hello World
>>>
```

You may also specify the classpath to your own jar files, or third-party jar files, when firing up the REPL. Then you can interactively use instances of your classes or third-party classes from the REPL as well.

Run as a script

You saw earlier how to compile Kotlin code into bytecode, create a jar file, and then run the code using either the java or the kotlin commands. This two-step process is useful when you have multiple files in a large application. But not everything we write is large or enterprise scale—shell scripts and batch files have their places.

To perform some back-end tasks, parse some files, copy files around based on some configuration—in other words, for things you'd typically use shell scripts—you may write a script using Kotlin. The benefit in doing so is that you don't have to remember the shell commands between sh, zsh, bash, Windows CMD, PowerShell, and so on. And you can use a powerful and fluent language to perform the tasks. Once you implement the desired task in Kotlin, you may run it as script, in a single step, instead of explicitly compiling the code to create bytecode.

If code has a syntax error, then the execution of the script will fail without actually running any part of the script; so it's pretty much as safe to run as script as it is to compile and execute.

Let's write a Kotlin script to list all files with a kts extension in the current directory. Here's the code for that:

```kotlin
java.io.File(".")
  .walk()
  .filter { file -> file.extension == "kts"}
  .forEach { println(it) }
```

The content of the file is not any different from a regular Kotlin file you'd write. The only difference is in the file name, the `kts` extension—to signify the intent to run as a script—instead of the `kt` extension.

The code uses the File class from the JDK `java.io` package, and also uses the extension functions that Kotlin has added to that class. It walks through all the files in the current (.) directory, filters or picks only files that end with the `kts` extension, and prints the file object—that is, the full path and name of each file picked.

To run this file, we'll use the `kotlinc-jvm` command, but this time instead of

To run this file, we'll use the `kotlinc-jvm` command, but this time instead of compiling, we'll ask the tool to run the code as script immediately. For this, use the `-script` option, like so:

```
kotlinc-jvm -script listktsfiles.kts
```

Here's the output from this code:

```
./listktsfiles.kts
```

On Unix-like systems, if you like to run the script without prefixing with `kotlinc-jvm -script`, then you may use the shebang facility, like this:

```
// greet.kts

#!/usr/bin/env kotlinc-jvm -script
 println("hi")
```

Make sure to run `chmod +x greet.kts` to give execution permission for the file.

Then, run the file directly from command line, like so:

```
./greet.kts
```

This will produce the output:

```
hi
```

On some systems you have to provide the full path to the location of `kotlinc-jvm` instead of `/usr/bin/env` for the shebang facility to work properly.

If you intend to use scripts in production, you may find script useful. It's a library that provides several capabilities to work with Kotlin scripts, including compiled script caching.

Kotlin code can not only be compiled to Java bytecode, it can also be compiled down to several other formats, as we'll see next.