Joint Compilation

We'll cover the following

- How to intermix code
- Running from the command line

How to intermix code

You can intermix code written using Java and Kotlin two different ways:

- Bring code written in either of those languages into a Java or Kotlin project as a JAR file dependency.
- Have source files written in the two languages, side by side, in a project.

The first approach is the most common. Currently you bring dependencies from Maven or JCenter into your Java projects using a build tool like Maven or Gradle. Likewise, you can bring dependencies into your Kotlin projects. The JAR files your code depends on may be created from Java, Kotlin, or both. If there are no compatibility issues, using these JAR files written in Kotlin or Java feels natural, just like using Java JAR files in Java projects. Any compatibility issues that may surface will be due to the language differences and not due to the JAR file integration. The techniques discussed later in this chapter will help resolve those issues.

The second approach, of mixing both Java and Kotlin source code in one project, is an option if you're introducing Kotlin into legacy Java projects and you want to make use of the power of Kotlin in some areas of the application.

Obviously, to compile the code written in the two different languages, you'll have to use the compilers of the respective languages. However, complications will arise from interdependencies.

Suppose your Kotlin code is calling some of your Java code, and some of your Java code is calling your Kotlin code. Compiling Java code first will fail since the Kotlin code it depends on hasn't been compiled yet into bytecode. Thankfully, this situation can be resolved by running the Kotlin compiler first and then the Java compiler. For this method to work, you must provide both the Kotlin source files and the Java source files to the Kotlin compiler. Upon seeing the Java source files, the Kotlin compiler will create stubs for the classes and methods in the given Java source files so that the dependencies of the Kotlin code are satisfied. Once the bytecode from the Kotlin code is generated, when you run the Java compiler, it will find the necessary Kotlin code dependencies for the Java code.

If you're using Maven or Gradle to run your build, then refer to the documentation to set up the project for joint compilation.

Irrespective of the tools you use to build, running the compilations from the command line will give a clear view of the underlying mechanism. For that reason, let's explore compiling code for a small sample project that mixes source files from both Java and Kotlin.

The following project contains two Kotlin source files under the directory `jointcompilation/src/main/kotlin/com/agiledeveloper/joint` and one Java source file under the directory `jointcompilation/src/main/java/com/agiledeveloper/joint`. The `Constants` class written in Kotlin doesn't have any dependencies, and we'll take a look at that first:

```
package com.agiledeveloper.joint

class Constants {
  val freezingPointInF = 32.0
}
```

Constants.kt

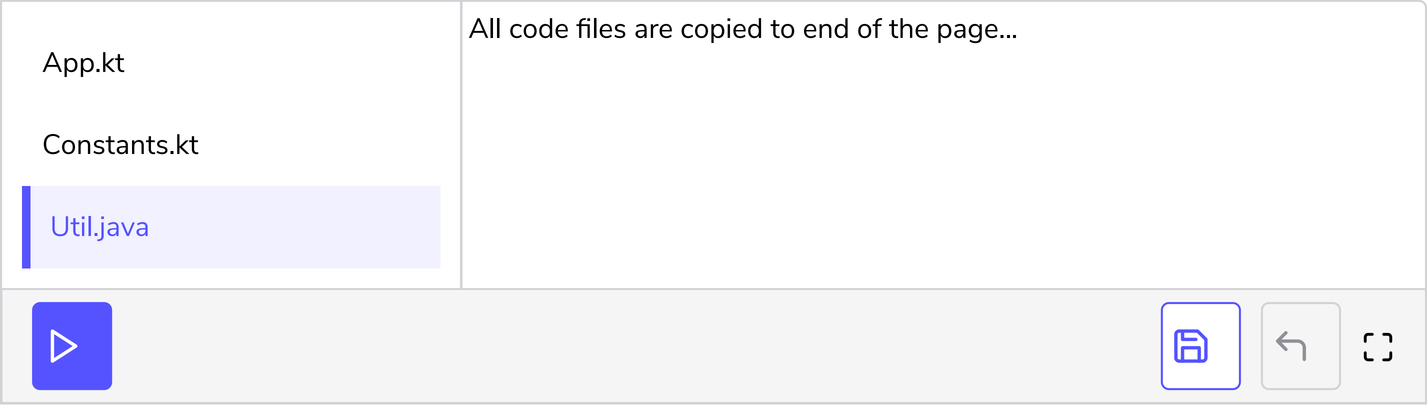
The `Constants` class has a property named `freezingPointInF`, which has been initialized to the expected value. Let's take a look at the Java class that will use this class:

```
package com.agiledeveloper.joint;

public class Util {
  public double f2c(double fahrenheit) {
    return (fahrenheit - new Constants().getFreezingPointInF()) * 5 / 9.0;
  }
}
```

Util.java

The Util Java class has a f2c() method that uses the Constants Kotlin class, but the syntax is no different from the way a Java class will use another Java class. The property in the Constants Kotlin class is accessed using a getter from Java, much like the way properties are accessed in Java. The Kotlin compiler creates getters for val properties and both getters and setters for var properties. Whereas we use the property name directly in Kotlin to access the properties of a class, from within Java we use the getters/setters. Let's look at some other Kotlin code that will make use of the Util Java class:



All code files are copied to end of the page...

App.kt

The App object, a singleton, has a main() method that has been marked with the JvmStatic annotation so the compiler will generate the method as a static method—we'll take a look at this annotation later in this chapter. Within the main() method, we use the Java class much like how we use any Kotlin class from within Kotlin.

Running from the command line

To compile this code locally, you have to first run the Kotlin compiler, but include both the Kotlin source files and the Java source file, like so:

```
kotlinc-jvm -d classes \src/main/kotlin/com/agiledeveloper/joint/*.kt \src/mai
n/java/com/agiledeveloper/joint/*.java
```

The command instructs the compiler to place the generated class files in the classes directory. If you take a peek into the classes/com/agiledeveloper/joint directory after running the above command, you'll see the files App.class and Constants.class. The compiler generated a stub for the Util Java class, but didn't save it into the destination directory. However, the compiler was able to use the stub to verify proper dependency of `App` on the `Util` class. And since the `Constants.class` file has been generated, the Java compiler should be able to verify the needs of the `Util` class. Let's now compile the `Util` Java class, using the Java compiler:

```
javac -d classes -classpath classes \src/main/java/com/agiledeveloper/joint/*.java
```

We provided the path to the `classes` directory as the classpath compile-time argument to the Java compiler. We specified the same directory as the destination directory for the generated `.class` file. After running the command, take a look at the `classes/com/agiledeveloper/joint` directory to confirm that the `.class` file exists for all the three files, two generated earlier by the Kotlin compiler and now the new one by the Java compiler.

You may run the code locally using the `kotlin` command or the `java` command. Let's run it using the `kotlin` command:

```
kotlin -classpath classes com.agiledeveloper.joint.App
```

All it needed was a reference to the classpath, to where the `.class` files are located.

To run it using the `java` command, include the path to the Kotlin standard library, like so:

```
java -classpath classes:$KOTLIN_PATH/lib/kotlin-stdlib.jar \ com.agiledeveloper.joint.App
```

On Windows, use `;` instead of `:` to separate the paths in the classpath and also `%KOTLIN_PATH%` for the environment variable that specifies the path to where Kotlin is installed on the system.

Whether you run the compiled code using `kotlin` or `java`, the output will be:

```
Running App...
10.0
```

If you're programming with modules for Java 9 or later, then place your code in the `modulepath` instead of the `classpath`.

In short, run the Kotlin compiler first and then the Java compiler. Also, remember to provide both Kotlin source files and Java source files to the Kotlin compiler so it can create the necessary stubs for Java code and verify that the Kotlin code's dependencies are correct.

The flexibility offered by the Kotlin compiler to create a stub for the Java code makes it possible to intermix Java and Kotlin source files in the same project. But that doesn't solve all the integration woes that may arise. Let's dive in to see some of the challenges that may arise when we use code written in one language in the other, irrespective of whether we use JAR dependencies or intermix source files from the two languages in the same project.

The next lesson explores how to call Java code from Kotlin.

Code Files Content !!!

⤋ ⤋

```
-------------------------------------------------------------------------
|  App.kt [1]
-------------------------------------------------------------------------



package com.agiledeveloper.joint

import kotlin.jvm.JvmStatic

object App {
  @JvmStatic
  fun main(@Suppress("UNUSED_PARAMETER") args: Array) {
    println("Running App...")

    println(Util().f2c(50.0))
  }
}



-------------------------------------------------------------------------
|  Constants.kt [1]
-------------------------------------------------------------------------
```

```
package com.agiledeveloper.joint

class Constants {
  val freezingPointInF = 32.0
}


-------------------------------------------------------------------------
|  Util.java [1]
-------------------------------------------------------------------------


package com.agiledeveloper.joint;

public class Util {
  public double f2c(double fahrenheit) {
    return (fahrenheit - new Constants().getFreezingPointInF()) * 5 / 9.0;
  }
}




**********************************************************************************
```