Calling Kotlin from Java

We'll cover the following

- Using overloaded operators from Java
- Creating static methods
- Passing lambdas
- Adding a throws clause
- Using functions with default arguments
- Accessing top-level functions
- More annotations

Once you compile Kotlin code to bytecode, you can use the `.class` files and JAR files created from Kotlin source files in Java projects. Alternatively, you may also intermix Kotlin source files, side by side, with Java source files and call Kotlin code from within Java code. We've already seen the mechanics to make this work. Let's now focus on the issues with source code we may run into with such integration efforts.

Kotlin has a number of features that don't exist in Java. Also, Kotlin is idiomatically different from Java in many ways. Thus, the Kotlin way of doing things that work elegantly and expressively when writing code in Kotlin won't work when calling Kotlin code from Java.

If the Kotlin code you're creating is for exclusive use with other Kotlin code, then don't worry about calling from Java—for example, with code related to UI or controllers and services written to run within a framework. Since no Java code will directly call such code, there's no reason to spend time and effort to make the code accessible from Java.

If you intend your Kotlin code to be used from Java code, then you have to take some extra steps for a smoother integration. If you're working on a project where such integration is needed, then integrate early and often to make sure that the Kotlin compiler is generating bytecode that's compatible with the needs of

Kotlin compiler is generating bytecode that's compatible with the needs of programmers using the code from Java. If you're creating a library in Kotlin for third-party programmers to use from Java, then create test code in Java, in addition to creating test code in Kotlin, and run those tests using continuous integration. This will help you to both verify that your code works as intended, whether it's called from Java or Kotlin, and also verify that your code integrates well when called from Java.

In spite of the idiomatic differences, the designers of the Kotlin language and the Kotlin standard library have provided many things to make the integration of Java code with Kotlin code as smooth as possible. When programming in Kotlin, we have to make use of some of these integration-related features so that our Kotlin code can be properly used from Java. Let's take a look at various features that are geared toward Java to Kotlin integration.

We'll create a `Counter` class in Kotlin, use it from a Kotlin script `usecounter.kts` and also from a `UseCounter` class written in Java. Using the `Counter` class from both Kotlin and Java will help us clearly see the extra steps we have to take for Java integration.

To practice along with the following examples, you'll have to compile the Kotlin code and the Java code separately. Take note of the following commands. As we incrementally develop the code, you can use these commands to compile and run the code, to see the outputs.

The following commands can be used to run the files locally:

To compile the Kotlin code, use the following command:

```
kotlinc-jvm -d classes src/main/kotlin/com/agiledeveloper/Counter.kt
```

When you're ready to execute the Kotlin script `usecounter.kts`, use this command:

```
kotlinc-jvm -classpath classes -script usecounter.kts
```

You can compile the Java code with this command:

```
javac -d classes -classpath classes:$KOTLIN_PATH/lib/kotlin-stdlib.jar \ src/m
ain/java/com/agiledeveloper/UseCounter.java
```

In addition to including the path to where the bytecode generated from Kotlin source is located, we also add the Kotlin standard library to the `classpath`. Later, as we work on the examples, we'll see why the Kotlin standard library is needed during compilation.

Finally, to run the Java class UseCounter, use the following command:

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

Now that we've seen the commands to compile, let's get down to the code. We'll walk through several integration issues and find ways to resolve each one of them.

Using overloaded operators from Java

When programming in Kotlin we can make use of operator overloading to create concise and expressive code. Operator-overloaded functions are created using well-defined method-naming conventions, as we saw in Overloading Operators. Let's create a `Counter` Kotlin data class with a `plus()` function to overload the `+` operator on instances of the class.

```
package com.agiledeveloper

data class Counter(val value: Int) {
    operator fun plus(other: Counter) = Counter(value + other.value)
}
```

Counter.kt

The `Counter` class's constructor takes an initial value of type `Int` for the `value` property. The class also has one method that will return a new `Counter` instance whose value is the sum of two `Counter` instances used as operands for the `+` operator.

Before we look at using this class from Java, let's use it from within a Kotlin script:

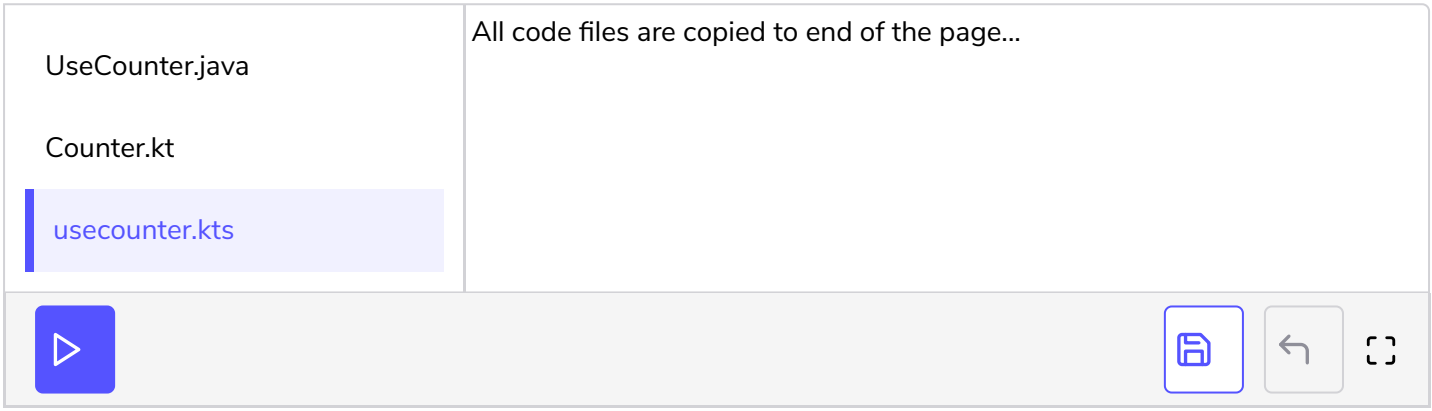
```
import com.agiledeveloper.*

val counter = Counter(1)
println(counter + counter)
```

usecounter.kts

We used the `+` operator to sum up the values in the two operands. This fluency of

Kotlin isn't possible in Java, since Java doesn't permit operator overloading. Not all is lost, though. The method-naming convention used in Kotlin for operators was chosen carefully to promote interoperability with Java. From within Java, instead of using `+` we can use the `plus()` function, like so:



Within the `main()` method of `UseCounter` Java class, we create an instance of `Counter` and invoke the `plus()` method on it. Compile the Kotlin code and then the Java code, using the commands mentioned previously, and execute the Java code. The output will display the class name, along with the value of the `value` property, as shown in the comment on the line with the call to `plus()` in the example code.

When programming in Kotlin, we don't have to do anything special to permit Java code to make use of operator overloading. Whereas the Kotlin code will use the operators, the Java code will use the corresponding methods.

Creating static methods

Kotlin doesn't have static methods. The closest to static methods in Kotlin are methods we create in singletons (Singleton with Object Declaration) and companion objects (Companion Objects and Class Members). We can call the methods in singleton objects and companion objects without creating an instance of a class in Kotlin. But to easily call these methods from Java, we have to instruct the Kotlin compiler to mark these methods as `static` in the bytecode. That can be achieved using the `JvmStatic` annotation. Let's add a companion object to the `Counter` class and mark a `create()` method in it with that annotation.

```
//within the Counter class...
companion object {
    @JvmStatic
    fun create() = Counter(0)
}
```

Counter.kt

For this code to compile correctly, we have to import the `JvmStatic` class in the top of the `Counter.kt` file, right after the package declaration line:

```
// Counter.kt
import kotlin.jvm.JvmStatic
```

Irrespective of whether we mark the methods in singletons or companion objects with the `JvmStatic` annotation, we can call the methods from Kotlin without explicitly creating an instance, as you can see in the following code to call the `create()` method of the `Counter` companion object:

```
// usecounter.kts
println(Counter.create())
```

However, since the `create()` method is marked with the annotation, we can call that method from Java as well, much like how we would call `static` methods in Java. Let's add the following code to the `main()` method of the `UseCounter` Java class.

```
// UseCounter.java

//within the main method of UseCounter...
Counter counter0 = Counter.create();
System.out.println(counter0); //Counter(value=0)
```

Try removing the `JvmStatic` annotation in the Kotlin code and notice how the Java code no longer compiles. When creating methods that belong to a singleton or a companion object, ask if you intend each one of those methods to be easily accessible as a `static` method from Java. If the answer to that question for a method is yes, then annotate it; otherwise leave out the annotation.

Passing lambdas

In addition to receiving objects, both in Kotlin and in Java, functions may receive other functions as arguments. On the receiving end, the parameters for the lambda expressions are backed by functional interfaces in Java, like `Runnable`, `Consumer<T>`, `Function<T, R>`, or your own homegrown interfaces with exactly one abstract method. In the case of Kotlin, functions that receive lambda expressions

are defined using a different syntax when compared to how such functions are defined in Java—see Receiving Lambdas. But, internally, Kotlin also uses functional interfaces to represent lambda expressions. Let's create a function in Kotlin that receives a lambda expression and see how we can make use of it in Java.

Let's add a method named `map()` to the Counter class in Kotlin. This method will receive a lambda expression as its parameter.

```
// Counter.kt
fun map(mapper: (Counter) -> Counter) = mapper(this)
```

The type of the `mapper` parameter of the `map()` function is a lambda expression—an anonymous function—that takes a `Counter` instance as its parameter and returns a `Counter` instance as its result. In the implementation of the `map()` function, we invoke the lambda expression referenced by the `mapper` parameter and pass the current object this as argument. The instance returned by the lambda expression is in turn returned by the `map()` function.

In Chapter 11, Functional Programming with Lambdas, we saw how to invoke higher-order functions—that is, functions that take lambda expressions. Let's call the above `map()` function from within Kotlin code and then compare that call to a call from Java.

```
// usecounter.kts
println(counter.map { ctr -> ctr + ctr })
```

Since the `map()` function takes only one parameter, and that happens to be a lambda expression, we can use the flexible Kotlin syntax with `{}` to pass the lambda expression. Java doesn't permit the use of `{}`, and lambdas are placed within `()`. Here's the Java code to call the `map()` method:

```
// UseCounter.java
System.out.println(counter.map(ctr -> ctr.plus(ctr))); //Counter(value=2)
```

If that works, then that syntax is pretty good—it means that we're able to pass a lambda expression written in Java to a function written in Kotlin that takes a lambda expression. But if you try to compile that Java code, you'll get an error saying `kotlin.jvm.functions.Function1` is not found. The reason for that error is, internally, the bytecode generated for the `map()` function references a functional interface defined in the Kotlin standard library. The Java compiler tries to bind the

interface defined in the Kotlin standard library. The Java compiler tries to bind the

lambda expression to that interface and complains that it has no knowledge of that interface. We can fix that easily by adding an import.

```
// UseCounter.java
import kotlin.jvm.functions.Function1;
```

The compiler needs access to the Kotlin standard library which contains the definition of the `Function1` interface. That's the reason—earlier, when we looked at the compilation commands—we included the Kotlin standard library in the `classpath` during compilation.

With the import in place, we can pass a lambda expression written in Java to the `map()` function. Depending on the number of arguments the lambda expressions take, we have to import interfaces like `Function0`, `Function1`, `Function2`, and so on, from the Kotlin standard library. A quick look at the error message from the Java compiler will reveal the details of what we need to import.

Adding a throws clause

Unlike the Java compiler which distinguishes between checked exceptions and unchecked exceptions, the Kotlin compiler treats them as one. We saw in try-catch Is Optional, how the Kotlin compiler doesn't force you to handle exceptions. You may choose to handle an exception in a particular function or let the exception propagate to the caller. The low ceremony, flexible nature of Kotlin allows us to evolve code more freely, as our understanding of the requirements improves. However, that flexibility will get in the way if we intend to access the functions written in Kotlin from Java. The reason is that the Java compiler won't allow you to place a `catch` for a checked exception unless the signature of the method being called has a `throws` clause. To illustrate this issue, and find a solution, let's write a function in Kotlin that may potentially result in an exception.

Add a `readFile()` method to the `Counter` class that will return the contents of a file whose path is given as the parameter.

```
// Counter.kt
fun readFile(path: String) = java.io.File(path).readLines()
```

The `readFile()` function is using the `java.io.File` class which may blow up with a

checked exception `java.io.FileNotFoundException` if the given path is invalid. The

Kotlin compiler doesn't force us to deal with the exception on the spot, just like the Java compiler doesn't force us to deal with unchecked exceptions.

When calling the `readFile()` function from within Kotlin, we may decide to play it safe and handle the exception, in case the call results in an exception. Let's make a call to the `readFile()` function but wrap it within a `try-catch`:

```
try {
    counter.readFile("blah")
} catch(ex: java.io.FileNotFoundException) {
    println("File not found")
}
```

usecounter.kts

Execute the Kotlin script and you'll notice the code reports a `File not found` error message since a file named `blah` doesn't exist in the current directory. If you don't handle the exception, the script will blow up.

Now let's turn our attention to calling the `readFile()` function from Java.

```
try {
    counter.readFile("blah");
} catch(java.io.FileNotFoundException ex) {
    System.out.println("File not found");
}
```

UseCounter.java

The above Java code is the syntactical equivalent of the code we wrote in Kotlin to call the `readFile()` function. But when we compile the Java code, we'll get an error:

```
exception FileNotFoundException is never thrown in body of corresponding try
  statement
```

The Java compiler notices that the catch targets a checked exception, but the `readFile()` function, in the generated bytecode, isn't marked with the `throws` clause. To place the call to `readFile()` in Java within a `try` block and to handle a checked exception, we'll have to tell the Kotlin compiler to generate the appropriate `throws` clause. This is done using the `Throws` annotation.

Revisit the Counter class and update the `readFile()` method definition to add the annotation.

```
// Counter.kt
@Throws(java.io.FileNotFoundException::class)
fun readFile(path: String) = java.io.File(path).readLines()
```

The annotation tells the Kotlin compiler to add a throws `java.io.FileNotFoundException` clause in the bytecode as part of the signature of the `readFile()` method.

After making the above change and recompiling the file `Counter.kt`, recompile the Java file `UseCounter.java` with the call to `readFile()` wrapped within the `try-catch` block. The code will compile with no errors this time, and when run, the Java code will print the same error message as the Kotlin version did.

Using functions with default arguments

Kotlin's default arguments feature that we saw in Default and Named Arguments, allows us to omit some arguments when making function calls. Let's explore how this feature pans out when we call functions with default arguments from Java.

Open the `Counter` class and add a new function `add()` that takes one parameter with a default argument value.

```
// Counter.kt
fun add(n: Int = 1) = Counter(value + n)
```

From within the Kotlin code, we can call the `add()` function, either with one argument of type `Int` or without any arguments. When called with no arguments, the Kotlin compiler will pass the default value of `1` to the `add()` function.

```
// usecounter.kts
println(counter.add(3))
println(counter.add())
```

That flexibility is nice. Let's try to make use of that from Java code.

```
// UseCounter.java
System.out.println(counter.add(3));
System.out.println(counter.add());
```

In the first line, we call the `add()` function with one argument, `3`. That's not a problem. If you want to call from Java a function with values for all the arguments, there's no issue. But if you like to take advantage of the default arguments and call, as in this example, the `add()` function without all the arguments, the compiler will choke up. The call to `add()` with no arguments will result in the following error:

```
error: method add in class Counter cannot be applied to given types;
```

The reason for this failure is that the Java compiler is complaining that it couldn't find a function `add()` that doesn't take any arguments.

If you'd like Java programmers to be able to call your functions with default arguments with fewer than all the possible arguments, then you can instruct the Kotlin compiler to generate overloaded functions using the `JvmOverloads` annotation. Upon seeing this annotation, the Kotlin compiler will generate multiple overloaded functions, as necessary, and provide a thin implementation in each to route the call to the version that takes all the parameters.

Let's modify the `add()` function in the `Counter` class to add the annotation.

```
// Counter.kt
@JvmOverloads
fun add(n: Int = 1) = Counter(value + n)
```

After this change, compile the Kotlin code and then the Java code. Both the calls to `add(3)` and `add()` will now compile.

Accessing top-level functions

Functions aren't required to belong to a class or a singleton in Kotlin. We saw in Chapter 4, Working with Functions, that top-level functions belong to packages. We can directly import the functions, from their respective packages, into Kotlin and use them without much fanfare. Let's see how to use the top-level functions from within Java code.

First, we'll create a top-level function in the `com.agiledeveloper` package, right above the `Counter` class in the file `Counter.kt`.

```
// Counter.kt

//place this after the import and before the data class Counter
```

```
fun createCounter() = Counter(0)
```

Using this `createCounter()` top-level function is easy. We already have the line import `com.agiledeveloper.*` in the script file `usecounter.kts`. So we can readily refer to the top-level function in the script:

```
// usecounter.kts
println(createCounter())
```

It'll take a bit more effort to access this top-level function from within Java. In the bytecode, top-level functions aren't permitted, and Kotlin's top-level functions have to be sheltered in a class. By default, Kotlin chooses to place the top-level functions in a class that's derived from the name of the file that contains the code.

In this example, the top-level function `createCounter()` is in the file `Counter.kt`. The Kotlin compiler decided to place that file in a class named `CounterKt`. Once you create a top-level function, you can see the `CounterKt.class` file being created in the `classes/com/agiledeveloper` directory when you compile the `Counter.kt` file using the Kotlin compiler command.

To access the top-level function, prefix the name of the function with the fully qualified class name, like so:

```
// UseCounter.java
System.out.println(com.agiledeveloper.CounterKt.createCounter());
```

If the generated class name doesn't please you, you can change the name of the class that will hold the top-level functions of your package using the Jvm- Name annotation. You may use this annotation to resolve function signature collisions, to change the names of getters or setters, and so on. In this example, we'll use the annotation to define the class name that this file will map to.

```
// Counter.kt
@file:JvmName("CounterTop")
package com.agiledeveloper
```

With the `@file:JvmName` annotation, placed before the `package` declaration, we tell the Kotlin compiler that the top-level function in this file—`Counter.kt`—should be in a class file `CounterTop.class` instead of in `CounterKt.class`. Compile the code and take a peek at the generated bytecode to confirm the new name of the `.class`

file.

Now we can change the Java code to use the new name we gave for the host of the top-level function in our package.

```
System.out.println(com.agiledeveloper.CounterTop.createCounter());
```

Compile the code and execute it to confirm that the Java to Kotlin integration works as expected.

More annotations

The annotations we've seen so far are useful for telling the Kotlin compiler to generate bytecode that allows the Java code to seamlessly interact with Kotlin code. The `kotlin.jvm` package has many more annotations to tailor different aspects of bytecode generation. For example, you can instruct the Kotlin compiler to mark a method as `synchronized` using the `Synchronized` annotation. You can instruct it to mark a method in an interface as a `default` method using the `JvmDefault` annotation. You can customize backing fields generated by the Kotlin compiler to be `volatile` by using the `Volatile` annotation or `transient` by using the `Transient` annotation.

Explore the different annotations available in the `kotlin.jvm` package, but don't go overboard with them. Use only annotations that are absolutely necessary, based on real use, and not for hypothetical extensibility.

The next lesson concludes the discussion for this chapter.

Code Files Content !!!

⤋ ⤋

```
------------------------------------------------------------------------
|  UseCounter.java [1]
------------------------------------------------------------------------
```

```
package com.agiledeveloper;
import kotlin.jvm.functions.Function1;


public class UseCounter {
    public static void main(String[] args) {
        Counter counter = new Counter(1);
        System.out.println(counter.plus(counter)); //Counter(value=2)
    }
}


------------------------------------------------------------------------
|  Counter.kt [1]
------------------------------------------------------------------------


package com.agiledeveloper

data class Counter(val value: Int) {
    operator fun plus(other: Counter) = Counter(value + other.value)
}


------------------------------------------------------------------------
|  usecounter.kts [1]
------------------------------------------------------------------------


import com.agiledeveloper.*

val counter = Counter(1)
println(counter + counter)




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