

Calling Java from Kotlin

We'll cover the following ^

- The Java code
- Running from the command line
 - Calling methods from Java

Calling code written in Java from within Kotlin `.kt` files and `.kts` scripts is straightforward for most part. Kotlin naturally integrates with Java, and we can use properties and methods without thinking twice—it almost always just works.

The Java code

To see this in action, and learn ways to work around occasional glitches you may run into, let's write a Java class that we will then use from Kotlin.

```
package com.agiledeveloper;

import java.util.List;
import static java.util.stream.Collectors.toList;

public class JavaClass {
    public int getZero() { return 0; }

    public List<String> convertToUpper(List<String> names) {
        return names.stream()
            .map(String::toUpperCase)
            .collect(toList());
    }

    public void suspend() {
        System.out.println("suspending...");
    }

    public String when() {
        return "Now!";
    }
}
```

JavaClass.java

The class `JavaClass` has a getter method, a `convertToUpper()` method, a method named `suspend()` and another method named `when()`. The purpose of the first

named `suspend()`, and another method named `when()`. The purpose of the first two methods is to illustrate how Kotlin works well with Java. The last two methods will help to see how Kotlin deals with Java method names that conflict with keywords in Kotlin.

Running from the command line

In this example, the Kotlin code we'll soon write will depend on Java code, but no Java code depends on Kotlin code. So we can straightaway compile the Java code using the Java compiler, instead of having to run the Kotlin compiler first. Here's the command to compile the Java code locally:

```
javac -d classes src/main/java/com/agiledeveloper/*
```

This command places the generated `.class` file under the `classes` subdirectory. We can use this from Kotlin code written in a `.kt` file or script written in a `.kts` file. Let's write a script in a file named `sample.kts`, which you can locally run using the following command:





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

Let's exercise the `getZero()` method of the `JavaClass` from within a Kotlin script written in a file named `sample.kts`.

sample.kts

JavaClass.java

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



sample.kts

We imported the Java class, created an instance of the class, and accessed the property `zero`. The fact that the class was written using Java didn't matter, and we used the class using the natural Kotlin syntax. Even though the Java class has a getter, we access the property from Kotlin using the name of the property instead of the getter. Run the script using the command mentioned previously to see the result `0` printed on the console.

Calling methods from Java

It was easy to call the property on the instance of `JavaClass`. Calling the `convertToUpper()` method from Kotlin shouldn't pose any challenges either.

Let's add a call to the `convertToUpper()` method to the `sample.kts` file.

```
println(javaObject.convertToUpper(listOf("Jack", "Jill"))) //[JACK, JILL]
```

To the `convertToUpper()` method we passed an instance of `List<String>`, created using the Kotlin's `listOf()` method. Since Kotlin's collections are compile-time views and map directly to JDK collections, there's no runtime overhead or compile-time stunts to use the Kotlin collections API to interact with Java code that uses the JDK collections. Run the script to verify that the output shows the list returned by the method call.

Let's step it up a notch to see how Kotlin handles a call to the `suspend()` method of the `JavaClass`. Kotlin's `suspend` keyword is used to mark functions as suspendible, but will the Kotlin compiler choke up on a call to a method named `suspend`? Let's address that question straight on to put an end to the suspense.

To the `sample.kts` file add a call to the `suspend()` method, like so:

```
javaObject.suspend() //suspending...
```

Execute the script to see if the compiler had any issues with the call. You'll see the output of the `suspend()` method on the console. Phew.

Kotlin handled that gracefully, even though `suspend` is a keyword. The compiler doesn't flinch, and the program executes the call to the `suspend()` method of the `JavaClass` just fine.

With the encouragement from the previous result, let's move ahead to calling the `when()` method of the `JavaClass`. In Kotlin, `when` is both an expression and a statement. Let's see if Kotlin is able to handle a call to a method named `when()`, much like how it was able to deal with `suspend()`.

```
println(javaObject.when()) //error: expecting an expression
```

As an experienced programmer, you know it's not a question of *if*, but of *when*

things fall apart. The Kotlin compiler chokes up when it sees the call to `when()`, and complains that it is expecting to see a `when` expression of Kotlin. A rogue developer who may be opposed to using Kotlin may now name methods `when()` just to make our lives difficult. Should we keep this quiet—what gives?

Thankfully, this isn't a showstopper to calling Java code from Kotlin. The language provides an escape facility, a work-around, for situations like this—the backtick.

When there's a keyword conflict between Java code and Kotlin, use the backtick operator to escape the method or property name:

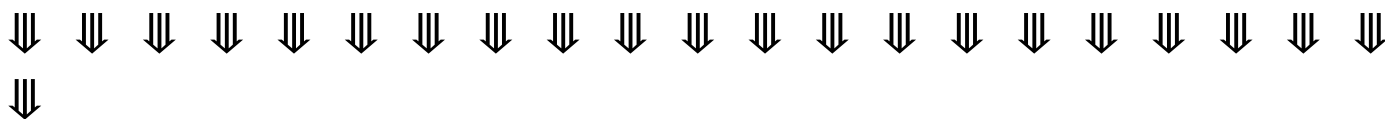
```
println(javaObject.`when`()) //Now!
```

When there's a conflict with a Kotlin keyword, we merely wrap the property or method name within a pair of backtick symbols, and off we go.

Kotlin was designed with interoperating with Java code and libraries in mind. Thus, calling Java code from Kotlin isn't an issue, and any minor glitches have an easy work-around.

In the next lesson, let's take a look at calling Kotlin code from within Java.

Code Files Content !!!



```
-----  
| sample.kts [1]  
-----
```

```
import com.agiledeveloper.JavaClass  
  
val javaObject = JavaClass()  
  
println(javaObject.zero) // 0
```

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

```
package com.agiledeveloper;

import java.util.List;
import static java.util.stream.Collectors.toList;

public class JavaClass {
    public int getZero() { return 0; }

    public List convertToUpper(List names) {
        return names.stream()
            .map(String::toUpperCase)
            .collect(toList());
    }

    public void suspend() {
        System.out.println("suspending...");
    }

    public String when() {
        return "Now!";
    }
}
```
