

Chapter 3

Scripting

JSR 223, Scripting for the Java Platform describes various mechanisms for allowing scripting language programs to access information in the Java platform and permitting scripting language pages to be used in a Java server-side application. The concept of enabling communication between scripts and program objects itself is not new. A notable example is how JavaScript, the scripting language supported by most browsers, can access methods in Java applets or Flash programs. A non-Java example: VBScript that can be used to access ActiveX objects inside Microsoft Office applications.

Note

The JSR 223 documentation can be downloaded from
<http://jcp.org/en/jsr/detail?id=223>

JSR 223 defines a standard on how to do this kind of communication with multiple script languages. Java 6 provides a script engine based on Rhino, an open source implementation of JavaScript. Rhino is written in Java and can be downloaded from <http://www.mozilla.org/rhino>. Support for other scripting languages can be found in the Scripting project at java.net (<https://scripting.dev.java.net>). The languages supported so far in this project include Groovy, Java, Jelly, Jexl, JudoScript, OGNL, Pnuts, Python, Ruby, Scheme, Sleep, Tcl, xpath, and XSLT. This chapter concentrates on the JavaScript engine in Java 6.

The Scripting API is defined and implemented as types in the **javax.script** package. This package offers the following areas of functionality.

1. Script execution. This feature allows Java programmers to run scripts written in a scripting language for which an engine is available. For Java 6, only JavaScript scripts are supported.

2. **Binding.** This unit of functionality enables Java programmers to access Java objects from script programs. Those Java objects must first be bound to variables.
3. **Compilation.** Before scripts can be executed, the corresponding script engine must first compile the script into intermediate code. This feature allows the storage of such intermediate code so that scripts that are invoked repeatedly need only be compiled once, hence speeding up the whole execution process.
4. **Invocation.** This feature is related to compilation. However, invocation enables intermediate code to be reused. The difference is very subtle. Compilation allows the whole script to be re-executed and invocation allows individual procedures/functions in the script to be re-executed.
5. **Script engine discovery and metadata.** For a scripting language to interact with Java, a script engine for that language must be available. The Scripting API allows script engines to be registered and discovered at run time. In addition, you can also query attributes about registered script engines.

The rest of this section takes a look at the core types in the **javax.script** package and provide several examples.

Core Types

The **javax.script** package contains six interfaces, five classes, and one exception. This section explains the more important types in this package.

The **ScriptEngineManager** Class

When working with the Scripting API, it is almost always the case that you need to first obtain the script engine object for the scripting language you're working with. The **ScriptEngine** interface models script engines, and the **ScriptEngineManager** class provides convenient methods to register and obtain a **ScriptEngine** object.

You can instantiate the **ScriptEngineManager** class by invoking one of its two constructors:

```
public ScriptEngineManager()
```

```
public ScriptEngineManager(java.lang.ClassLoader loader)
```

The first constructor uses the default class loader, which is the one obtained by calling `Thread.currentThread().getContextClassLoader()`. The second constructor allows you to use a different class loader than the default.

To register a script engine, you call the `registerEngineExtension`, `registerEngineMimeType`, or `registerEngineName` methods on a `ScriptEngineManager` object. You do this if you have a script engine implementation of your own that you want to expose. Unless you're developing a script engine, this is something you will very rarely do. Most often, you will use `ScriptEngineManager` to obtain a script engine.

To obtain a script engine, you call the `getEngineByName` method on the `ScriptEngineManager` object. Its signature is as follows.

```
public ScriptEngine getEngineByName(java.lang.String shortName)
```

Since Rhino is the only script engine coming with Java 6, you use the `getEngineByName` method by passing the short name for JavaScript: "js". Therefore,

```
ScriptEngineManager manager = new ScriptEngineManager();  
ScriptEngine jsEngine = manager.getEngineByName("js");
```

There is also another interesting method, `getEngineFactories`, that returns a list of `ScriptEngineFactory` objects. Its signature is this.

```
public java.util.List<ScriptEngineFactory> getEngineFactories()
```

The example in the section "Listing All Script Engines" lists all script engines in the current implementation of Java. It will be more interesting when Java includes more engines in the future. For now, be content with Rhino.

The ScriptEngineFactory Interface

A `ScriptEngineFactory` object encapsulates metadata that describes a script engine. Therefore, implementing a script engine also requires writing an implementation of `ScriptEngineFactory`. The following are the more important methods of the `ScriptEngineFactory` interface.

```
public java.lang.String getEngineName()
```

Returns the full name of the script engine.

```
public java.lang.String getEngineVersion()
```

Return the version number of the script engine.

```
public java.lang.String getLanguageName()
```

Returns the name of the supported scripting language.

```
public java.lang.String getLanguageVersion()
```

Returns the version of the supported scripting language.

```
public java.util.List<java.lang.String> getNames()
```

Returns an immutable list of short names that identify the supported script engine.

```
public java.lang.Object getParameter(java.lang.String key)
```

Returns the value of the specified attribute.

```
public ScriptEngine getScriptEngine()
```

Returns an instance of the supported script engine.

The Bindings Interface

I introduce this interface before the **ScriptEngine** interface, which is a much more important interface than **Bindings**, because **ScriptEngine** uses **Bindings**. As such, understanding of **Bindings** is crucial to learning **ScriptEngine**.

Bindings is a subinterface of **java.util.Map** and **Bindings** objects can be used to store key/value pairs that may be useful when executing scripts. For example, you can store a key/value pair in a **Bindings** instance and expose it to scripts. The scripts can then access the key/value pair as if it were a global variable.

Here are the methods defined in the **Bindings** interface:

```
public boolean containsKey(java.lang.Object key)
```

Indicates whether or not this **Bindings** contains the specified key.

```
public java.lang.Object get(java.lang.Object key)
```

Returns the value bound to the specified key.

```
public java.lang.Object put(java.lang.String key,  
    java.lang.Object value)
```

Binds the specified key to the specified value. If there is already a key with the specified name, replace the value with the specified value. This

method returns the value previously associated with the specified key. It returns null if no value was previously mapped to the key.

```
public void putAll(java.util.Map<? extends java.lang.String,  
    ? extends java.lang.Object> map)
```

Adds the contents of the specified **Map** to this **Bindings**.

```
public java.lang.Object remove(java.lang.Object key)
```

Removes the key/value pair for the specified key from the **Bindings** and returns the removed value, if any.

The ScriptEngine Interface

An instance of the **ScriptEngine** interface represents a script engine. The most useful methods in this interface can be used to execute scripts and create bindings. The **eval** method, which comes in several overloads, is used to execute scripts. The **put** method can be used to bind keys with values. Here are the signatures of the more important methods.

```
public java.lang.Object eval(java.lang.String script)  
    throws ScriptException
```

Executes the specified script. This method returns the value resulting from the execution of the script.

```
public java.lang.Object eval(java.io.Reader reader)  
    throws ScriptException
```

Executes the script in *reader* and returns the result of the execution.

```
public java.lang.Object eval(java.lang.String script,  
    Bindings bindings)
```

Executes the script and provides a **Bindings** that can be accessed from the script.

```
public java.lang.Object eval(java.io.Reader reader,  
    Bindings bindings)
```

Executes the script in *reader* and provides a **Bindings** that can be accessed from the script.

```
public java.lang.Object eval(java.lang.String script,  
    ScriptContext context)
```

Executes the script and provides access to the specified **ScriptContext**.

```
public java.lang.Object eval(java.io.Reader reader,  
    ScriptContext context)
```

Executes the script in *reader* and provides access to the specified **ScriptContext**.

```
public void put (java.lang.String key, java.lang.Object value)
    Binds the key with the value.
```

```
public java.lang.Object get (java.lang.String key)
    Returns the value bound to the specified key.
```

```
public ScriptEngineFactory getFactory ()
    Returns the ScriptEngineFactory object that encapsulates metadata for
    this ScriptEngine instance.
```

Listing All Script Engines

The code in Listing 3.1 shows how to list all script engine factories that come with Java 6.

Listing 3.1: Listing script engine factories

```
import java.util.List;
import javax.script.ScriptEngineManager;
import javax.script.ScriptEngineFactory;

public class ListEngineFactoryDemo {

    public static void main (String[] args) {
        // create ScriptEngineManager
        ScriptEngineManager manager = new ScriptEngineManager ();
        List<ScriptEngineFactory> factoryList =
            manager.getEngineFactories ();
        for (ScriptEngineFactory factory : factoryList) {
            System.out.println (factory.getEngineName ());
            System.out.println (factory.getLanguageName ());
        }
    }
}
```

If you compile and run this class, you will see the following result:

```
Mozilla Rhino
ECMAScript
```

Running Scripts

The code in Listing 3.2 shows how you can run JavaScript code from inside a Java program.

Listing 3.2: RunScriptDemo class

```
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class RunScriptDemo {

    public static void main(String[] args) {
        // create a ScriptEngineManager object
        ScriptEngineManager manager = new ScriptEngineManager();
        // get the ScriptEngine
        ScriptEngine engine = manager.getEngineByName("js");

        // execute script
        String script = "print('hello')";
        try {
            engine.eval(script);
        } catch (ScriptException e) {
            e.printStackTrace();
        }
    }
}
```

Run the **RunScriptDemo** class and you will see “hello” on your console.

As another example, the code in Listing 3.3 shows the **RunScriptFileDemo** class that can be used to execute Javascript script in a file.

Listing 3.3: RunScriptFileDemo class

```
import java.io.FileReader;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;

public class RunScriptFileDemo {
    public static void main(String[] args) {
        // the first argument must be the path to the script file
    }
}
```

```
    if (args.length != 1) {
        System.out.println(
            "Usage: java RunScriptFile [file]");
        System.exit(0);
    }

    // create ScriptEngineManager
    ScriptEngineManager manager = new ScriptEngineManager();

    // get the ScriptEngine for JavaScript (js)
    ScriptEngine engine = manager.getEngineByName("js");

    // open file, and execute the script
    try {
        FileReader reader = new FileReader(args[0]);
        engine.eval(reader);
        reader.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Listing 3.4 shows a file named **sample.js** that contains a Javascript function **add** and a script that invokes the function.

Listing 3.4: The sample.js file

```
function add(a, b) {
    print('Adding ' + a + ' with ' + b + ' ...');
    c = a + b;
    return c;
}

result = add(10, 5);
print('Result = ' + result);
```

You can run the **RunScriptFileDemo** class by running this command line command.

```
java RunScriptFileDemo sample.js
```

Note that the **sample.js** file must be located in the same directory as the directory you are invoking the command. Otherwise, you must pass the whole path to the script file.

This is the result from running the Java class:

```
Adding 10 with 5 ...  
Result = 15
```

Binding Scripts

As yet another example, Listing 3.5 shows the **BindingDemo** class that illustrates the use of variables that are bound through **ScriptEngine**, so that the variables can be used for the next execution of the script.

Listing 3.5: The **BindingDemo** class

```
import javax.script.Bindings;  
import javax.script.ScriptContext;  
import javax.script.ScriptEngine;  
import javax.script.ScriptEngineManager;  
import javax.script.ScriptException;  
  
public class BindingDemo {  
  
    public static void main(String[] args) {  
  
        // create ScriptEngineManager  
        ScriptEngineManager manager = new ScriptEngineManager();  
  
        // get the ScriptEngine for JavaScript  
        ScriptEngine engine = manager.getEngineByName("js");  
  
        // bind a to 10 and b to 5  
        engine.put("a", 10);  
        engine.put("b", 5);  
  
        // get bound values  
        Bindings bindings = engine.getBindings(  
            ScriptContext.ENGINE_SCOPE);  
        Object a = bindings.get("a");  
        Object b = bindings.get("b");  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
  
        // use the bound values in calculations  
        try {
```

```
        Object result = engine.eval("c = a + b;");
        System.out.println("a + b = " + result);
    } catch (ScriptException e) {
        e.printStackTrace();
    }
}
```

Note that we bound **a** and **b** in these lines of code:

```
engine.put("a", 10);
engine.put("b", 5);
```

And then, we used it in a JavaScript script:

```
Object result = engine.eval("c = a + b;");
```

The output from the **BindingDemo** class is as follows:

```
a = 10
b = 5
a + b = 15.0
```

Using Invocable

Any script that needs executing will first have to be compiled into intermediate code. This compilation takes a relatively large amount of CPU cycles. The same procedure or function that gets called twice will have to be compiled twice too. With the **Invocable** interface you can save time by storing the compiled intermediate code for reuse if the same procedure or function is called the second time.

An implementation of **ScriptEngine** can optionally implement the **Invocable** interface. Since implementing this interface is optional, you need to first perform an **instanceof** check before upcasting a **ScriptEngine** to this interface.

```
if (scriptEngine instanceof Invocable) {
    Invocable invocable = (Invocable) scriptEngine;
    // invoke method/procedure here
}
```

The **Invocable** interface provides the **invoke** methods to invoke a scripting function.

```
public java.lang.Object invoke(java.lang.String functionName,
    Object... args)
    throws ScriptException, java.lang.NoSuchMethodException
```

Invokes the specified top-level function. Note that you can pass any number of arguments.

```
public java.lang.Object invoke(java.lang.Object instance,
    java.lang.String functionName, Object... args)
    throws ScriptException, java.lang.NoSuchMethodException
```

Invokes the specified function on *instance*. Note that you can pass any number of arguments.

In addition, the **Invocable** interface provides two other methods:

```
public <T> T getInterface(java.lang.Class<T> clazz)
```

Returns the implementation of an interface using compiled procedures.

```
public <T> T getInterface(java.lang.Object instance,
    java.lang.Class<T> clazz)
```

Returns the implementation of an interface using compiled procedures. The argument *instance* refers to the scripting object whose member functions are used to implement the methods of the interface.

As an example, the **InvocableDemo** in Listing 3.6 illustrates the use of **Invocable** through which you can call procedures that have been compiled in subsequent executions of a script. You can call a procedure using one of these methods:

- Use the **invoke** method define in the **Invocable** interface
- Use your own interface, in which case the implementation of this custom interface comes from the **getInterface** method.

In the example, we'll use the custom **Adder** interface in Listing 3.7.

Listing 3.6: The **InvocableDemo** class

```
import javax.script.Invocable;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;

public class InvocableDemo {
```

```
public static void main(String[] args) {

    // create ScriptEngineManager
    ScriptEngineManager manager = new ScriptEngineManager();

    // get the ScriptEngine for JavaScript (js)
    ScriptEngine engine = manager.getEngineByName("js");

    try {
        // execute script that contains the add function
        engine.eval("function add(a, b) " +
            "{c = a + b; return c;}");
        Invocable jsInvoke = (Invocable) engine;

        // call the add function, passing 10 and 5
        Object result1 = jsInvoke.invoke("add",
            new Object[] {10, 5});
        System.out.println(result1);

        // call the add function through the add method
        // in the Adder interface
        Adder adder = jsInvoke.getInterface(Adder.class);
        int result2 = adder.add(10, 5);
        System.out.println(result2);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Listing 3.7: The Adder interface

```
public interface Adder {
    int add(int a, int b);
}
```

Running the **InvocableDemo** class produces the following output on the console:

```
15.0
15
```

Using Compilable

The **Compilable** interface is similar to **Invocable**. However, instead of storing the intermediate code of a specific function/procedure, with **Compilable** you store the intermediate code of an entire script. Like **Invocable**, **Compilable** is an optional interface that is implemented by a **ScriptEngine** implementation, thereby performing an **instanceof** check is necessary before upcasting an instance of **ScriptEngine** to **Compilable**.

```
if (scriptEngine instanceof Compilable) {
    Compilable compilable = (Compilable) scriptEngine
    // invoke script here
}
```

The **Compilable** interface provides two overloads of the **compile** method:

```
public CompiledScript compile(java.lang.String script)
    throws ScriptException
    Compiles the script for later execution.

public CompiledScript compile(java.io.Reader reader)
    throws ScriptException
    Compiles the script in reader for later execution.
```

The return value of the **compile** method is a **CompiledScript** object. You can then execute the script by calling one of the **eval** methods on the **CompiledScript** object. Here are the three overloads of **eval**, which are similar to those defined in the **ScriptEngine** interface:

```
public java.lang.Object eval()
public java.lang.Object eval(Bindings bindings)
public java.lang.Object eval(ScriptContext context)
```

Listing 3.8 shows a class (**CompilableDemo**) that illustrates the use of the **Compilable** interface.

Listing 3.8: The CompilableDemo Class

```
import javax.script.Compilable;
import javax.script.CompiledScript;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
```

```
public class CompilableDemo {

    public static void main(String[] args) {

        // create ScriptEngineManager
        ScriptEngineManager manager = new ScriptEngineManager();

        // create ScriptEngine for JavaScript (js)
        ScriptEngine engine = manager.getEngineByName("js");

        Compilable jsCompile = (Compilable) engine;
        try {
            CompiledScript script = jsCompile.compile(
                "function hi() {print('Hello !'); }; hi();");

            for (int i = 0; i < 5; i++) {
                script.eval();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Running the class in Listing 3.8 generates the following message on the console.

```
Hello !
Hello !
Hello !
Hello !
Hello !
```

Summary

JSR 223, Scripting for the Java Platform enables collaboration between Java and scripting languages. Java 6 comes with a reference implementation that supports JavaScript, however other scripting languages are supported through the Scripting project. This chapter presents the API and teaches how to use them.