CHAPTER 9

Scripting in Java

QUESTIONS AND EXERCISES

1. What is a scripting language?

Answer:

A *scripting language* is a programming language that provides the ability to write *scripts* that are evaluated (or interpreted) by a runtime environment called a *script engine* (or an interpreter). A script is a sequence of characters that is written using the syntax of a scripting language and used as the source for a program executed by an interpreter. The interpreter parses the scripts, produces intermediate code, which is an internal representation of the program, and executes the intermediate code. The interpreter stores the variables used in a script in data structures called *symbol tables*.

Typically, unlike in a compiled programming language, the source code (called a script) in a scripting language is not compiled, but is interpreted at runtime. However, scripts written in some scripting languages may be compiled into Java bytecode that can be run by the JVM.

2. What JDK module contains the Scripting API?

Answer:

The `java.scripting` module contains the Scripting API.

3. What is Nashorn JavaScript?

Answer:

Nashorn JavaScript is Oracle's implementation of the ECMAScript language. The code written Nashorn JavaScript can be run by the JVM.

4. What is the name of the script engine that is co-bundled with JDK9?

Answer:

The name of the script engine that is co-bundled with JDK9 is Nashorn JavaScript.

5. Briefly describe the use of the following classes and interfaces: `ScriptEngineFactory`, `ScriptEngine`, `ScriptEngineManager`, `Compilable`, `Invocable`, `Bindings`, `ScriptContext`, and `ScriptException`.

Answer:

A `ScriptEngineFactory` performs two tasks: It creates instances of the script engine and It provides information about the script engine such as engine name, version, language, etc.

The `ScriptEngine` interface is the main interface in the Java Scripting API whose instances facilitate the execution of scripts written in a particular scripting language.

The `ScriptEngineManager` class provides a discovery and instantiation mechanism for script engines. It also maintains a mapping of key-value pairs as an instance of the `Bindings` interface storing state that is shared by all script engines that it creates.

The `Compilable` interface may optionally be implemented by a script engine that allows compiling scripts for their repeated execution without recompilation.

The `Invocable` interface may optionally be implemented by a script engine that may allow invoking procedures, functions, and methods in scripts that have been compiled previously.

An instance of a class that implements the `Bindings` interface is a mapping of key-value pairs with a restriction that a key must be non-null, non-empty `String`. It extends the `java.util.Map` interface. The `SimpleBindings` class is an implementation of the `Bindings` interface.

An instance of the `ScriptContext` interface acts as a bridge between the Java host application and the script engine. It is used to pass the execution context of the Java host application to the script engine. The script engine may use the context information while executing a script. A script engine may store its state in an instance of a class that implements the ScriptContext interface, which may be accessible to the Java host application.

The ScriptException class is an exception class. A script engine throws a ScriptException if an error occurs during the execution, compilation, or invocation of a script. The class contains three useful methods called getLineNumber(), getColumnNumber(), and getFileName(). These methods report the line number, the column number, and the file name of the script in which the error occurs. The ScriptException class overrides the getMessage() method of the Throwable class and includes the line number, column number, and the file name in the message that it returns.

6. What is the use of the eval() method of a ScriptEngine?

Answer:

The eval() method of a ScriptEngine is used to evaluate a script.

7. Write a program in which you create an instance of the ScriptContext interface using SimpleScriptContext class. Store a few attributes in the engine scope and global scope, retrieve the same attributes, and print their values.

Solution:

```
// ScriptContextTest.java
package com.jdojo.script.exercises;

import javax.script.Bindings;
import javax.script.ScriptContext;
import javax.script.SimpleBindings;
import javax.script.SimpleScriptContext;
import static javax.script.ScriptContext.ENGINE_SCOPE;
import static javax.script.ScriptContext.GLOBAL_SCOPE;

public class ScriptContextTest {
    public static void main(String[] args) {
        // Create a script context
        ScriptContext ctx = new SimpleScriptContext();

        // Store the attributes
        ctx.setAttribute("day", 22, ENGINE_SCOPE);


        // Add a global scope Bindings to the context. By default, it is null
        Bindings globalBindings = new SimpleBindings();
        ctx.setBindings(globalBindings, GLOBAL_SCOPE);
        ctx.setAttribute("year", 2018, GLOBAL_SCOPE);

        // Retrieve the attributes
        Integer day = (Integer) ctx.getAttribute("day");
        Integer year = (Integer) ctx.getAttribute("year", GLOBAL_SCOPE);

        System.out.println("ENGINE_SCOPE, day = " + day);
        System.out.println("GLOBAL_SCOPE, year = " + year);
    }
}
```

8. How do you add attributes to the global scope and engine scope?

Answer:

Use the `put(String key, Object value)` method of the `ScriptEngineManager` class to add attributes to the global scope.

Use the `put(String key, Object value)` of the `ScriptEngine` interface to add attributes to the engine scope.

9. How do you send the output of scripts executed by a `ScriptEngine` to a file?

Answer:

Use the `setWriter(Writer writer)` method of the `ScriptContext` of the `ScriptEngine` to write the script output to a file. The following snippet of code sets `soutput.txt` file in the current directory as the script output destination:

```
ScriptEngine engine = /* Get a script engine */;
FileWriter writer = new FileWriter("jsoutput.txt");
ScriptContext defaultCtx = engine.getContext();
defaultCtx.setWriter(writer);
```

This example sets a custom output writer for the default context of the `ScriptEngine` that will be used during the execution of scripts that use the default context. If you want to use a custom output writer for a specific execution of a script, you need to use a custom `ScriptContext` and set its writer.

10. Write a snippet of code that checks if a `ScriptEngine` supports compiling scripts.

Answer:

```
ScriptEngine engine = /* Get the script engine */;

if (engine instanceof Compilable) {
    System.out.println("Script compilation is supported.");
} else {
    System.out.println("Script compilation is not supported.");

}
```

11. Write part of the `java` command in which you enable ES6 support in Nashorn.

Answer:

```
-Dnashorn.args=--language=es6
```

12. How do you import Java types into Nashorn JavaScript using the `Packages` object and `Java.type()` function?

Answer:

Nashorn defines all Java packages as properties of a global variable named `Packages`. For example, the `java.lang` and `javax.swing` packages may be referred to as `Packages.java.lang` and `Packages.javax.swing`, respectively. The following snippet of code uses the `java.util.List` and `javax.swing.JFrame` in Nashorn:

```
// Create a List
var list1 = new Packages.java.util.ArrayList();

// Create a JFrame
var frame1 = new Packages.javax.swing.JFrame("Test");
```

Nashorn also declares `java`, `javax`, `org`, `com`, `edu`, and `net` as global variables that are aliases for `Packages.java`, `Packages.javax`, `Packages.org`, `Packages.com`, `Packages.edu`, and `Packages.net`, respectively.

The `type()` function of the `Java` object imports a Java type into the script. You need to pass the fully qualified name of the Java type to import. In Nashorn, the following snippet of code imports the `java.util.ArrayList` class and creates its object:

```
// Import java.util.ArrayList type and call it ArrayList
var ArrayList = Java.type("java.util.ArrayList");
```

```
// Create an object of the ArrayList type
var list = new ArrayList();
```

In the code, you call the imported type returned from the `Java.type()` function as `ArrayList` that is also the name of the class that is imported. You do it to make the next statement read as if it was written in Java. Readers of the second statement will know that you are creating an object of the `ArrayList` class. However, you can give the imported type any name you want. The following snippet of code imports `java.util.ArrayList` and calls it `MyList`:

```
// Import java.util.ArrayList type and call it MyList
var MyList = Java.type("java.util.ArrayList");

// Create an object of the MyList type
var list2 = new MyList();
```

13. Create an unmodifiable list of two strings using the `of()` method of the `java.util.List` interface and print the values in the list. Use Nashorn JavaScript to write the code.

Solution:

```
var List = Java.type("java.util.List");
var list = List.of(10, 20, 30);
print(list);
```

14. Create an array of `int`. Add two elements to the array with values 100 and 300. Print the elements in the array - one element on a single line. Use Nashorn JavaScript to write the code.

Solution:

```
var intArray = java.lang.reflect.Array.newInstance(java.lang.Integer.TYPE, 2);
intArray[0] = 100;
intArray[1] = 300;
for(var i = 0; i < intArray.length; i++) {
    print(intArray[i]);
}
```

15. If you want to roll out your own script engine, what is the name of the service interface whose implementation you must provide?

Answer:

`javax.script.ScriptEngineFactory`

16. What are the `jrunscript` and `jjs` tools?

    Answer:

    The `jrunscript` tool is a command-line script shell. It is script-engine-neutral and it can be used to evaluate any script including Nashorn JavaScript..You can find this tool in the `JAVA_HOME\bin` directory.

    The `jjs` tool is a command-line tool to work with the Nashorn script engine, The tool is located in the `JDK_HOME\bin` directory. The tool can be used to run scripts in files or scripts entered on the command-line in interactive mode. It can also be used to execute shell scripts.

17. How do you start the `jjs` tool, so you can execute shell scripts?

    Answer:

    Use the `-scripting` option with the `jjs` tool , so the tool can execute shell scripts.

18. How do you exit out of the `jjs` tool?

    Answer:

    Execute the `quit()` function to exit the `jjs` tool.