Java Platform, Standard Edition

Java Scripting Programmer's Guide

Release 13
F18345-01
September 2019

ORACLE

Java Platform, Standard Edition Java Scripting Programmer's Guide, Release 13

F18345-01

Contents

Preface

This document provides an overview of the scripting features in Java Platform, Standard Edition (Java SE).

Audience

This document is intended for Java application programmers who want to execute code written in scripting languages. It is assumed that the you are familiar with Java and a version of the ECMAScript language standard (JavaScript).

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

See the Java Scripting API specification, the `javax.script` package.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1
Scripting Languages and Java

This section describes the characteristics of scripting languages and how they can be used by Java programmers.
Scripting languages are programming languages that support the ability to write scripts. Unlike source files for other programming languages that must be compiled into bytecode before you run them, scripts are evaluated by a runtime environment (in this case, by a script engine) directly.

Most scripting languages are dynamically typed. This enables you to create new variables without declaring the variable type (the interpreter assigns the type based on the type of the object associated with the variable), and you can reuse the same variable for objects of different types (type conversion is performed automatically).

Scripting languages generally have simple syntax; they allow complex tasks to be performed in relatively few steps.

Although scripting languages are usually interpreted at runtime, they can be compiled into Java bytecode that can then be executed on the Java Virtual Machine (JVM). Scripting languages can be faster and easier to use for certain problems, so it is sometimes chosen by developers of Java applications. However, if you write your Java application in a scripting language, then you lose the benefits of the Java language (such as type safety and access to the class library).

Java Specification Request (JSR) 223: Scripting for the Java Platform addresses the issue of integrating Java and scripting languages. It defines a standard framework and application programming interface (API) to embed scripts in your Java applications and access Java objects from scripts.

By embedding scripts in your Java code, you can customize and extend the Java application. For example, you can have configuration parameters, business logic, math expressions, and other external parts written as scripts. When developing your Java application, you do not need to choose the scripting language. If you write your application with the Java Scripting API (defined by JSR 223), then users can write scripts in any language compliant with JSR 223. See The Java Scripting API.

When writing a script in a language compliant with JSR 223, you have access to the entire standard Java library. See Using Java from Scripts.

2
The Java Scripting API

This section introduces the Java Scripting API and describes how the Java Scripting API (defined by JSR 223) is used to embed scripts in your Java applications. It also provides a number of examples with Java classes, which demonstrate the features of the Java Scripting API.

> **Note:**
>
> The Nashorn engine is deprecated in JDK 11 in preparation for removal in a future release.

Topics

The JavaScript Package

The Java Scripting API consists of classes and interfaces from the `javax.script` package. It is a relatively small and simple package with the `ScriptEngineManager` class as the starting point. A `ScriptEngineManager` object can discover script engines through the JAR file service discovery mechanism, and instantiate `ScriptEngine` objects that interpret scripts written in a specific scripting language.

The Nashorn engine is the default ECMAScript (JavaScript) engine bundled with the Java SE Development Kit (JDK). The Nashorn engine was developed fully in Java by Oracle as part of an OpenJDK project, Project Nashorn.

Although Nashorn is the default ECMAScript engine used by the Java Scripting API, you can use any script engine compliant with JSR 223, or you can implement your own. This document does not cover the implementation of script engines compliant with JSR 223, but at the most basic level, you must implement the `javax.script.ScriptEngine` and `javax.script.ScriptEngineFactory` interfaces. The abstract class `javax.script.AbstractScriptEngine` provides useful defaults for a few methods in the `ScriptEngine` interface.

How to Use the Java Scripting API to Embed Scripts

To use the Java Scripting API:

1. Create a `ScriptEngineManager` object.
2. Get a `ScriptEngine` object from the manager.
3. Evaluate the script using the script engine's `eval()` method.

Java Scripting API Examples with Java Classes

The following examples show you how to use the Java Scripting API in Java. To keep the examples simple, exceptions are not handled. However, there are checked and runtime exceptions thrown by the Java Scripting API, and they should be properly handled. In every example, an instance of the `ScriptEngineManager` class is used to request the Nashorn engine (an object of the `ScriptEngine` class) using the `getEngineByName()` method. If the engine with the specified name is not present, `null` is returned. For more information about using the Nashorn engine, see the *Nashorn User's Guide*.

> **Note:**
>
> Each `ScriptEngine` object has its own variable scope; see Using Multiple Scopes.

Evaluating a Statement

In this example, the `eval()` method is called on the script engine instance to execute JavaScript code from a `String` object.

```
import javax.script.*;

public class EvalScript {
    public static void main(String[] args) throws Exception {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("nashorn");

        // evaluate JavaScript code
        engine.eval("print('Hello, World')");
    }
}
```

Evaluating a Script File

In this example, the `eval()` method takes in a `FileReader` object that reads JavaScript code from a file named `script.js`. By wrapping various input stream objects as readers, it is possible to execute scripts from files, URLs, and other resources.

```
import javax.script.*;

public class EvalFile {
    public static void main(String[] args) throws Exception {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("nashorn");

        // evaluate JavaScript code
        engine.eval(new java.io.FileReader("script.js"));
    }
}
```

Exposing a Java Object as a Global Variable

In this example, a `File` object is created and exposed to the engine as a global variable named `file` using the `put()` method. Then the `eval()` method is called with JavaScript code that accesses the variable and calls the `getAbsolutePath()` method.

> **Note:**
>
> The syntax to access fields and call methods of Java objects exposed as variables depends on the scripting language. This example uses JavaScript syntax, which is similar to Java.

```
import javax.script.*;
import java.io.*;

public class ScriptVars {
    public static void main(String[] args) throws Exception {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("nashorn");

        // create File object
        File f = new File("test.txt");

        // expose File object as a global variable to the engine
        engine.put("file", f);

        // evaluate JavaScript code and access the variable
        engine.eval("print(file.getAbsolutePath())");
    }
}
```

Invoking a Script Function

In this example, the `eval()` method is called with JavaScript code that defines a function with one parameter. Then, an `Invocable` object is created and its `invokeFunction()` method is used to invoke the function.

> **Note:**
>
> Not all script engines implement the `Invocable` interface. This example uses the Nashorn engine, which can invoke functions in scripts that have previously been evaluated by this engine.

```
import javax.script.*;

public class InvokeScriptFunction {
    public static void main(String[] args) throws Exception {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("nashorn");

        // evaluate JavaScript code that defines a function with one parameter
        engine.eval("function hello(name) { print('Hello, ' + name) }");
```

```
        // create an Invocable object by casting the script engine object
        Invocable inv = (Invocable) engine;

        // invoke the function named "hello" with "Scripting!" as the argument
        inv.invokeFunction("hello", "Scripting!");
    }
}
```

Invoking a Script Object's Method

In this example, the `eval()` method is called with JavaScript code that defines an object with a method. This object is then exposed from the script to the Java application using the script engine's `get()` method. Then, an `Invocable` object is created, and its `invokeMethod()` method is used to invoke the method defined for the script object.

> **Note:**
>
> Not all script engines implement the `Invocable` interface. This example uses the Nashorn engine, which can invoke methods in scripts that have previously been evaluated by this engine.

```
import javax.script.*;

public class InvokeScriptMethod {
    public static void main(String[] args) throws Exception {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("nashorn");

        // evaluate JavaScript code that defines an object with one method
        engine.eval("var obj = new Object()");
        engine.eval("obj.hello = function(name) { print('Hello, ' + name) }");

        // expose object defined in the script to the Java application
        Object obj = engine.get("obj");

        // create an Invocable object by casting the script engine object
        Invocable inv = (Invocable) engine;

        // invoke the method named "hello" on the object defined in the script
        // with "Script Method!" as the argument
        inv.invokeMethod(obj, "hello", "Script Method!");
    }
}
```

Implementing a Java Interface with Script Functions

In this example, the `eval()` method is called with JavaScript code that defines a function. Then, an `Invocable` object is created, and its `getInterface()` method is used to create a `Runnable` interface object. The methods of the interface are implemented by script functions with matching names (in this case, the `run()` function is used to implement the `run()` method in the interface object). Finally, a new thread is started that runs the script function.

```
import javax.script.*;

public class ImplementRunnable {
    public static void main(String[] args) throws Exception {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("nashorn");

        // evaluate JavaScript code that defines a function with one parameter
        engine.eval("function run() { print('run() function called') }");

        // create an Invocable object by casting the script engine object
        Invocable inv = (Invocable) engine;

        // get Runnable interface object
        Runnable r = inv.getInterface(Runnable.class);

        // start a new thread that runs the script
        Thread th = new Thread(r);
        th.start();
        th.join();
    }
}
```

Implementing a Java Interface with the Script Object's Methods

In this example, the `eval()` method is called with JavaScript code that defines an object with a method. This object is then exposed from the script to the Java application using the script engine's `get()` method. Then, an `Invocable` object is created, and its `getInterface()` method is used to create a `Runnable` interface object. The methods of the interface are implemented by the script object's methods with matching names (in this case, the `run` method of the `obj` object is used to implement the `run()` method in the interface object). Finally, a new thread is started that runs the script object's method.

```
import javax.script.*;

public class ImplementRunnableObject {
    public static void main(String[] args) throws Exception {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("nashorn");

        // evaluate JavaScript code that defines a function with one parameter
        engine.eval("var obj = new Object()")
        engine.eval("obj.run = function() { print('obj.run() method called') }");

        // expose object defined in the script to the Java application
        Object obj = engine.get("obj");

        // create an Invocable object by casting the script engine object
        Invocable inv = (Invocable) engine;

        // get Runnable interface object
        Runnable r = inv.getInterface(obj, Runnable.class);

        // start a new thread that runs the script
        Thread th = new Thread(r);
        th.start();
        th.join();
    }
}
```

Using Multiple Scopes

In this example, the script engine's `put()` method is used to set the variable `x` to a `String` object `hello`. Then, the `eval()` method is used to print the variable in the default scope. Then, a different script context is defined, and its scope is used to set the same variable to a different value (a `String` object `world`). Finally, the variable is printed in the new script context that displays a different value.

A single scope is an instance of the `javax.script.Bindings` interface. This interface is derived from the `java.util.Map<String, Object>` interface. A scope is a set of name and value pairs where the name is a non-empty, non-null `String` object. The `javax.script.ScriptContext` interface supports multiple scopes with associated `Bindings` for each scope. By default, every script engine has a default script context. The default script context has at least one scope represented by the static field `ENGINE_SCOPE`. Various scopes supported by a script context are available through the `getScopes()` method.

```
import javax.script.*;

public class MultipleScopes {
    public static void main(String[] args) throws Exception {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("nashorn");

        // set global variable
        engine.put("x","hello");

        // evaluate JavaScript code that prints the variable (x = "hello")
        engine.eval("print(x)");

        // define a different script context
        ScriptContext newContext = new SimpleScriptContext();
        newContext.setBindings(engine.createBindings(), ScriptContext.ENGINE_SCOPE);
        Bindings engineScope = newContext.getBindings(ScriptContext.ENGINE_SCOPE);

        // set the variable to a different value in another scope
        engineScope.put("x", "world");

        // evaluate the same code but in a different script context (x = "world")
        engine.eval("print(x)", newContext);
```

3
Using Java from Scripts

This section describes how to access Java classes and interfaces from scripts.

> **Note:**
>
> The Nashorn engine is deprecated in JDK 11 in preparation for removal in a future release.

The code snippets are written in JavaScript, but you can use any scripting language compliant with JSR 223. Examples can be used as script files, or can be run in an interactive shell one expression at a time. The syntax for accessing fields and methods of objects is the same in JavaScript as it is in Java.

Topics

Accessing Java Classes

To access primitive and reference Java types from JavaScript, call the `Java.type()` function, which returns a type object that corresponds to the full name of the class passed in as a string. The following example shows you how to get various type objects:

```
var ArrayList = Java.type("java.util.ArrayList");
var intType = Java.type("int");
var StringArrayType = Java.type("java.lang.String[]");
var int2DArrayType = Java.type("int[][]");
```

The type object returned by the `Java.type()` function can be used in JavaScript code similar to how a class name is used in Java. For example, you can can use it to instantiate new objects as follows:

```
var anArrayList = new Java.type("java.util.ArrayList");
```

Java type objects can be used to instantiate new Java objects. The following example shows you how to instantiate new objects using the default constructor and by passing arguments to another constructor:

```
var ArrayList = Java.type("java.util.ArrayList");
var defaultSizeArrayList = new ArrayList;
var customSizeArrayList = new ArrayList(16);
```

You can use the type object returned by the `Java.type()` function to access static fields and methods as follows:

```
var File = Java.type("java.io.File");
File.createTempFile("nashorn", ".tmp");
```

To access a static inner class, use the dollar sign ($) in the argument passed to the `Java.type()` method. The following example shows how to return the type object of the `Float` inner class in `java.awt.geom.Arc2D`:

```
var Float = Java.type("java.awt.geom.Arc2D$Float");
```

If you already have the outer class type object, then you can access the inner class as a property of the outer class as follows:

```
var Arc2D = Java.type("java.awt.geom.Arc2D")
var Float = Arc2D.Float
```

In case of a nonstatic inner class, you must pass an instance of the outer class as the first argument to the constructor.

Although a type object in JavaScript is used similar to the Java class, it is distinct from the `java.lang.Class` object, which is returned by the `getClass()` method. You can obtain one from the other using the `class` and `static` properties. The following example shows this distinction:

```
var ArrayList = Java.type("java.util.ArrayList");
var a = new ArrayList;

// All of the following are true:
print("Type acts as target of instanceof: " + (a instanceof ArrayList));
print("Class doesn't act as target of instanceof: " + !(a instanceof a.getClass()));
print("Type is not the same as instance's getClass(): " + (a.getClass() !==
ArrayList));
print("Type's `class` property is the same as instance's getClass(): " +
(a.getClass() === ArrayList.class));
print("Type is the same as the `static` property of the instance's getClass(): " +
(a.getClass().static === ArrayList));
```

Syntactically and semantically, this distinction between compile-time class expressions and runtime class objects makes JavaScript similar to Java code. However, there is no equivalent of the `static` property for a `Class` object in Java, because compile-time class expressions are never expressed as objects.

Importing Java Packages and Classes

To access Java classes by their simple names, you can use the `importPackage()` and `importClass()` functions to import Java packages and classes. These functions are built into the compatibility script (`mozilla_compat.js`).

> **Note:**
>
> Avoid importing packages in production application code. Instead, import classes with the `importClass()` function and save them as global variables. Limit the use of the `importPackage()` function to simple prototyping with Java classes.

The following example shows you how to use the `importPackage()` and `importClass()` functions:

```
// Load compatibility script
load("nashorn:mozilla_compat.js");
// Import the java.awt package
importPackage(java.awt);
// Import the java.awt.Frame class
importClass(java.awt.Frame);
// Create a new Frame object
var frame = new java.awt.Frame("hello");
// Call the setVisible() method
frame.setVisible(true);
// Access a JavaBean property
print(frame.title);
```

You can access Java packages using the `Packages` global variable (for example, `Packages.java.util.Vector` or `Packages.javax.swing.JFrame`), but standard Java SE packages have shortcuts (`java` for `Packages.java`, `javax` for `Packages.javax`, and `org` for `Packages.org`).

The `java.lang` package is not imported by default, because its classes would conflict with `Object`, `Boolean`, `Math`, and other built-in JavaScript objects. Furthermore, importing any Java package or class can lead to conflicts with the global variable scope in JavaScript. To avoid this, define a `JavaImporter` object and use the `with` statement to limit the scope of the imported Java packages and classes, as shown in the following example:

```
// Create a JavaImporter object with specified packages and classes to import
var Gui = new JavaImporter(java.awt, javax.swing);

// Pass the JavaImporter object to the "with" statement and access the classes
// from the imported packages by their simple names within the statement's body
with (Gui) {
    var awtframe = new Frame("AWT Frame");
    var jframe = new JFrame("Swing JFrame");
};
```

Using Java Arrays

To create a Java array object, you first have to get the Java array type object, and then instantiate it. The syntax for accessing array elements and the `length` property in JavaScript is the same as in Java, as shown in the following example:

```
var StringArray = Java.type("java.lang.String[]");
var a = new StringArray(5);

// Set the value of the first element
a[0] = "Scripting is great!";
```

```
// Print the length of the array
print(a.length);
// Print the value of the first element
print(a[0]);
```

Given a JavaScript array, you can convert it to a Java array using the `Java.to()` method. You must pass the JavaScript array variable to this method and the type of array to be returned, either as a string or a type object. You can also omit the array type argument to return an `Object[]` array. Conversion is performed according to the ECMAScript conversion rules. The following example shows you how to convert a JavaScript array to a Java array using various `Java.to()` method arguments:

```
// Create a JavaScript array
var anArray = [1, "13", false];

// Convert the JavaScript array to a Java int[] array
var javaIntArray = Java.to(anArray, "int[]");
print(javaIntArray[0]); // prints the number 1
print(javaIntArray[1]); // prints the number 13
print(javaIntArray[2]); // prints the number 0

// Convert the JavaScript array to a Java String[] array
var javaStringArray = Java.to(anArray, Java.type("java.lang.String[]"));
print(javaStringArray[0]); // prints the string "1"
print(javaStringArray[1]); // prints the string "13"
print(javaStringArray[2]); // prints the string "false"

// Convert the JavaScript array to a Java Object[] array
var javaObjectArray = Java.to(anArray);
print(javaObjectArray[0]); // prints the number 1
print(javaObjectArray[1]); // prints the string "13"
print(javaObjectArray[2]); // prints the boolean value "false"
```

Given a Java array, you can convert it to a JavaScript array using the `Java.from()` method. The following example shows you how to convert a Java array that contains a list of files in the current directory to a JavaScript array with the same contents:

```
// Get the Java File type object
var File = Java.type("java.io.File");
// Create a Java array of File objects
var listCurDir = new File(".").listFiles();
// Convert the Java array to a JavaScript array
var jsList = Java.from(listCurDir);
// Print the JavaScript array
print(jsList);
```

> **Note:**
>
> In most cases, you can use the Java array in scripts without converting the Java array to a JavaScript array.

Implementing Java Interfaces

The syntax for implementing a Java interface in JavaScript is similar to how anonymous classes are declared in Java. You instantiate an interface and implement

its methods (as JavaScript functions) in the same expression. The following example shows you how to implement the `Runnable` interface:

```
// Create an object that implements the Runnable interface by implementing
// the run() method as a JavaScript function
var r  = new java.lang.Runnable() {
    run: function() {
        print("running...\n");
    }
};

// The r variable can be passed to Java methods that expect an object implementing
// the java.lang.Runnable interface
var th = new java.lang.Thread(r);
th.start();
th.join();
```

If a method expects an object that implements an interface with only one method, you can pass a script function to this method instead of the object. For instance, in the previous example, the `Thread()` constructor expects an object that implements the `Runnable` interface, which defines only one method. You can take advantage of automatic conversion and pass a script function to the `Thread()` constructor instead of the object. The following example shows you how you can create a `Thread` object without implementing the `Runnable` interface:

```
// Define a JavaScript function
function func() {
    print("I am func!");
};

// Pass the JavaScript function instead of an object that implements
// the java.lang.Runnable interface
var th = new java.lang.Thread(func);
th.start();
th.join();
```

You can implement multiple interfaces in a subclass by passing the relevant type objects to the `Java.extend()` function; see Extending Concrete Java Classes.

Extending Abstract Java Classes

You can instantiate an anonymous subclass of an abstract Java class by passing to its constructor a JavaScript object with properties whose values are functions that implement the abstract methods. If a method is overloaded, then the JavaScript function will provide implementations for all variants of the method. The following example shows you how to instantiate a subclass of the abstract `TimerTask` class:

```
var TimerTask =  Java.type("java.util.TimerTask");
var task = new TimerTask({ run: function() { print("Hello World!") } });
```

Instead of invoking the constructor and passing an argument to it, you can provide the argument directly after the `new` expression. The following example shows you how this syntax (similar to Java anonymous inner class definition) can simplify the second line in the previous example:

```
var task = new TimerTask {
    run: function() {
        print("Hello World!")
```

```
    }
};
```

If the abstract class is a functional interface (it has a single abstract method), then instead of passing a JavaScript object to the constructor, you can pass the function that implements the method. The following example shows how you can simplify the syntax when using a functional interface:

```
var task = new TimerTask(function() { print("Hello World!") });
```

Whichever syntax you choose, if you need to invoke a constructor with arguments, you can specify the arguments before the implementation object or function.

If you want to invoke a Java method that takes a functional interface as the argument, you can pass a JavaScript function to the method. Nashorn will instantiate a subclass of the expected class and use the function to implement its only abstract method. The following example shows you how to invoke the `Timer.schedule()` method, which expects a `TimerTask` object as the argument:

```
var Timer = Java.type("java.util.Timer");
Timer.schedule(function() { print("Hello World!") });
```

> **Note:**
>
> The previous syntax assumes that the expected functional interface is either an interface or it has a default constructor, which is used by Nashorn to instantiate a subclass of the expected class. It is not possible to use a non-default constructor.

Extending Concrete Java Classes

To avoid ambiguity, the syntax for extending abstract classes is not allowed for concrete classes. Because a concrete class can be instantiated, such syntax may be interpreted as an attempt to create a new instance of the class and pass to it an object of the type expected by the constructor (in case when the expected object type is an interface). As an illustration of this, consider the following example:

```
var t = new java.lang.Thread({ run: function() { print("Thread running!") } });
```

This code can be interpreted both as extending the `Thread` class with the specified implementation of the `run()` method, and the instantiation of the `Thread` class by passing to its constructor an object that implenents the `Runnable` interface. See Implementing Java Interfaces.

To extend a concrete Java class, pass its type object to the `Java.extend()` function that returns a type object of the subclass. Then, use the type object of the subclass as a JavaScript-to-Java adapter to create instances of the subclass with the specified method implementations. The following example shows you how to extend the `Thread` class with the specified implementation of the `run()` method:

```
var Thread = Java.type("java.lang.Thread");
var threadExtender = Java.extend(Thread);
var t = new threadExtender() {
    run: function() { print("Thread running!") }};
```

The `Java.extend()` function can take a list of multiple type objects. You can specify no more than one type object of a Java class, and as many type objects of Java interfaces as you want. The returned type object extends the specified class (or `java.lang.Object` if no class is specified) and implements all interfaces. The class type object does not have to be first in the list.

Accessing Methods of a Superclass

To access methods in the superclass, you can use the `Java.super()` function. Example 3-1 shows you how to extend the `java.lang.Exception` class and access the methods in the superclass.

If you run the code in Example 3-1, the following will be printed to standard output:

```
jdk.nashorn.javaadapters.java.lang.Exception: MY EXCEPTION MESSAGE
```

Example 3-1 Accessing Methods of a Supreclass (super.js)

```
var Exception = Java.type("java.lang.Exception");
var ExceptionAdapter = Java.extend(Exception);

var exception = new ExceptionAdapter("My Exception Message") {
    getMessage: function() {
        var _super_ = Java.super(exception);
        return _super_.getMessage().toUpperCase();
    }
}

try {
    throw exception;
} catch (ex) {
    print(exception);
}
```

Binding Implementations to Classes

The previous sections described how to extend Java classes and implement interfaces using an extra JavaScript object parameter in the constructor that specifies the implementation. The implementation is therefore bound to the actual instance created with `new`, and not to the whole class. This has some advantages, for example, in the memory footprint of the runtime, because Nashorn can create a single universal adapter for every combination of types being implemented. However, the following example shows that different instances have the same Java class regardless of them having different JavaScript implementation objects:

```
var Runnable = java.lang.Runnable;
var r1 = new Runnable(function() { print("I'm runnable 1!") });
var r2 = new Runnable(function() { print("I'm runnable 2!") });
r1.run();
r2.run();
print("We share the same class: " + (r1.class === r2.class));
```

The previous example prints the following:

```
I'm runnable 1!
I'm runnable 2!
We share the same class: true
```

If you want to pass the class for instantiation to an external API (for example, when using the JavaFX framework, the `Application` class is passed to the JavaFX API, which instantiates it), you must extend a Java class or implement an interface with the implementation bound to the class, rather than to its instances. You can bind the implementation to the class by passing a JavaScript object with the implementation as the last argument to the `Java.extend()` function. This creates a class with the same constructors as the original class, because they do not need an extra implementation object parameter. The following example shows you how to bind implementations to the class, and demonstrates that in this case the implementation classes for different invocations are different:

```
var RunnableImpl1 = Java.extend(java.lang.Runnable, function() { print("I'm runnable
1!") });
var RunnableImpl2 = Java.extend(java.lang.Runnable, function() { print("I'm runnable
2!") });
var r1 = new RunnableImpl1();var r2 = new RunnableImpl2();
r1.run();
r2.run();
print("We share the same class: " + (r1.class === r2.class));
```

The previous example prints the following:

```
I'm runnable 1!
I'm runnable 2!
We share the same class: false
```

Moving the implementation objects from the constructor invocations to the invocations of the `Java.extend()` functions eliminates the need for an extra argument in the constructor invocations. Every invocation of the `Java.extend()` function with a class-specific implementation object produces a new Java adapter class.The adapter classes with class-bound implementations can still take an additional constructor argument to further override the behavior for certain instances. Thus, you can combine the two approaches: you can provide part of the implementation in a class-based JavaScript implementation object passed to the `Java.extend()` function, and provide implementations for instances in objects passed to the constructor. A function defined by the object passed to the constructer overrides the function defined by the class-bound object. The following example shows you how to override the function defined in the class-bound object with a function passed to the constructor:

```
var RunnableImpl = Java.extend(java.lang.Runnable, function() { print("I'm runnable
1!") });
var r1 = new RunnableImpl();
var r2 = new RunnableImpl(function() { print("I'm runnable 2!") });
r1.run();
r2.run();
print("We share the same class: " + (r1.class === r2.class));
```

The previous example prints the following:

```
I'm runnable 1!
I'm runnable 2!
We share the same class: true
```

Selecting Method Overload Variant

Java methods can be overloaded by argument types. The Java Compiler (`javac`) selects the correct method variant during compilation. Overload resolution for Java methods called from Nashorn is performed when the method is invoked. The correct

variant is selected automatically based on the argument types. However, if you run into genuine ambiguity with actual argument types, you can specify a particular overload variant explicitly. This may also improve performance, because the Nashorn engine will not need to perform overload resolution during invocation.

Overload variants are exposed as special properties. You can refer to them in the form of strings that contain the name of the method followed by the argument types within parantheses. The following example shows how to invoke the variant of the `System.out.println()` method that expects an `Object` class as the argument, and pass `"hello"` to it:

```
var out = java.lang.System.out;
out["println(Object)"]("hello");
```

In the previous example, the unqualified class name (`Object`) is sufficient, because it uniquely identifies the correct signature. The only case when you must use the fully qualified class names in the signature is when two overload variants use different parameter types with identical unqualified names (this is possible if parameter types with the same name are from different packages).

Mapping Data Types

Most conversions between Java and JavaScript work as you expect. Previous sections described some of the less evident data type mappings between Java and JavaScript. Arays are automatically converted to Java array types such as `java.util.List`, `java.util.Collection`, `java.util.Queue`, and `java.util.Deque`. JavaScript functions are automatically converted to SAM types when they are passed as parameters to Java methods. Every JavaScript object implements the `java.util.Map` interface to enable APIs to receive maps directly. When numbers are passed to a Java API, they are converted to the expected target numeric type, either boxed or primitive. However, if the target type is less specific (for example, `Number`), you can only expect them to be of type `Number`, and must test specifically for whether the type is a boxed `Double`, `Integer`, `Long`, and so on. The number can be any boxed type due to internal optimizations. Also, you can pass any JavaScript value to a Java API expecting either a boxed or primitive number, because the `ToNumber` conversion algorithm defined by the JavaScript specification will be applied to the value. If a Java method expects a `String` or a `Boolean` object, the values will be converted using all conversions allowed by the `ToString` and `ToBoolean` conversions defined by the JavaScript specification. Nashorn ensures that internal JavaScript strings are converted to `java.lang.String` when exposed externally.

Passing JSON Objects to Java

The function `Java.asJSONCompatible(obj)` accepts a script object and returns an object that is compatible with the expectations of most Java JSON libraries: it exposes all arrays as `List` objects (rather than `Map` objects) and all other objects as `Map` objects.

Mapping Data Types mentions that every JavaScript object, when exposed to Java APIs, implements the `java.util.Map` interface. This is true even for JavaScript arrays. However, this behavior is often not desired or expected when Java code expects JSON-parsed objects. Java libraries that manipulate JSON-parsed objects usually expect arrays to expose the `java.util.List` interface instead. If you need to expose your JavaScript objects in such a manner that arrays are exposed as lists and not

maps, use the `Java.asJSONCompatible(obj)` function, where `obj` is the root of your JSON object tree.

Example 3-2 Example of Java.asJSONCompatible() Function

The following example calls the functions `JSON.parse()` and `Java.asJSONCompatible()` on the same JSON object. The function `JSON.parse()` parses the array `[2,4,5]` as a map while the function `Java.asJSONCompatible()` parses the same array as a list.

```
import javax.script.*;
import java.util.*;

public class JSONTest {
  public static void main(String[] args) throws Exception {
    ScriptEngineManager m = new ScriptEngineManager();
    ScriptEngine e = m.getEngineByName("nashorn");

    Object obj1 = e.eval(
      "JSON.parse('{ \"x\": 343, \"y\": \"hello\", \"z\": [2,4,5] }');");
    Map<String, Object> map1 = (Map<String, Object>)obj1;
    System.out.println(map1.get("x"));
    System.out.println(map1.get("y"));
    System.out.println(map1.get("z"));
    Map<Object, Object> array1 = (Map<Object, Object>)map1.get("z");
    array1.forEach((a, b) -> System.out.println("z[" + a + "] = " + b));

    System.out.println();

    Object obj2 = e.eval(
      "Java.asJSONCompatible({ \"x\": 343, \"y\": \"hello\", \"z\":
[2,4,5] })");
    Map<String, Object> map2 = (Map<String, Object>)obj2;
    System.out.println(map2.get("x"));
    System.out.println(map2.get("y"));
    System.out.println(map2.get("z"));
    List<Object> array2 = (List<Object>)map2.get("z");
    array2.forEach(a -> System.out.println(a));
  }
}
```

This example prints the following:

```
343
hello
[object Array]
z[0] = 2
z[1] = 4
z[2] = 5

343
hello
[2, 4, 5]
2
```

```
4
5
```