APPENDIX

B

Scripting API and Dynamically Typed Language Support

JavaScript, Ruby, PHP, and other scripting languages are popular choices for developing web-based (and other kinds of) applications. Recognizing their importance, Sun Microsystems added the Scripting API to Java 6. Scripting lets you develop applications that are partly based on Java and partly based on scripting languages. This appendix introduces you to the Scripting API, which Java 7 supports.

■ **Note** Wikipedia's "Scripting language" entry (`http://en.wikipedia.org/wiki/Scripting_language`), states that a *scripting language* is a "programming language that allows control of one or more applications" and makes the compiler of the language part of the language runtime, and as a result, enables code to be generated dynamically. "'Scripts" are distinct from the core code of the application, as they are usually written in a different language and are often created or at least modified by the end-user. Scripts are often interpreted from source code or bytecode, whereas the application is typically first compiled to native machine code."

If you're wondering why you might want to use a scripting language, consider that various scripting languages offer language features such as closures that aren't available in Java 7. Also, popular scripting language libraries (e.g., the JavaScript-based jQuery library [`http://en.wikipedia.org/wiki/JQuery`]) can simplify website development and more.

Although its name suggests otherwise, JavaScript isn't Java, and this name doesn't imply any technical resemblance. Originally known as LiveScript, the name JavaScript was chosen as a marketing decision.

Java 7 also supports dynamically typed languages via a new `invokedynamic` Java Virtual Machine (JVM) instruction, and a new dynamic linkage mechanism that's accessible via the types found in the new `java.lang.invoke` package and a pair of new types in the `java.lang` package – `BootstrapMethodError` and `ClassValue`. This appendix also introduces you to Java 7's support for dynamically typed languages.

■ **Note** A *dynamically typed language* performs most of its type checking at runtime instead of at compile time. With dynamic typing, values have types but variables do not. In other words, a variable can refer to a value of any type. In contrast, a *statically typed language* performs type checking at compile time.

Scripting API

Created under "JSR 223: Scripting for the Java Platform" (`http://www.jcp.org/en/jsr/detail?id=223`), the Scripting API lets Java applications and *scripts* written in different scripting languages interact with each other. This API is usable for both web and non-web applications, but this section focuses on the latter use.

The Scripting API is assigned the `javax.script` package. This package contains six interfaces, five regular classes, and one exception class, which collectively define *script engines* (software components that execute scripts specified as scripting-language-based source code) and provide a framework for using them in Java applications. Table B-1 describes `javax.script`'s classes and interfaces.

Table B-1. *Scripting API Classes and Interfaces*

Class/Interface	Description
`AbstractScriptEngine`	A class that abstracts a script engine via overloaded `eval()` methods.
`Bindings`	An interface that describes a mapping of key/value pairs, where keys are specified as `java.lang.String` instances.
`Compilable`	An interface that describes a script engine that lets scripts be compiled to intermediate code. A script engine class optionally implements `Compilable`.
`CompiledScript`	An abstract class extended by subclasses that store compilation results.
`Invocable`	An interface describing a script engine that lets a script's global functions and object member functions be invoked directly from Java code. It also lets scripts implement Java interfaces and Java code invoke script functions through those interfaces. A script engine class optionally implements `Invocable`.
`ScriptContext`	An interface used to connect script engines with scopes, which determine which script engines have access to various sets of key/value pairs. `ScriptContext` also exposes a reader and writers that script engines use for input/output operations.
`ScriptEngine`	An interface that represents a script engine. It provides methods to evaluate scripts, set and obtain script variables, and perform other tasks.

`ScriptEngineFactory`	An interface that describes and instantiates script engines. It provides methods that expose script engine metadata, such as the engine's version number.
`ScriptEngineManager`	A class that's the entry point into the Scripting API. It discovers and instantiates script engine factories, providing a method that lets an application enumerate these factories and retrieve a script engine that exposes the appropriate metadata (e.g., the correct language name and version number) from a factory. It also provides various methods for obtaining script engines by extension, Multipurpose Internet Mail Extensions (MIME) type, or short name. This class maintains a global scope; this scope's key/value pairs are available to all script engines created by the script engine manager.
`ScriptException`	A class that describes syntax errors and other problems occuring during script execution. Class members store the line number and column position where a problem occurred, and also the name of the file containing the script that was executing. This information's availability depends on the context in which the problem occurred. For example, a `ScriptException` instance thrown from executing a script that's not based on a file is unlikely to record a filename.
`SimpleBindings`	A class that provides a simple implementation of `Bindings`, which is backed by some kind of `java.util.Map` implementation.
`SimpleScriptContext`	A class that provides a simple implementation of `ScriptContext`.

As well as providing `javax.script` and its classes and interfaces, Java offers a script engine that understands JavaScript. This script engine is based on the Mozilla Rhino JavaScript implementation. Check out Mozilla's Rhino: JavaScript for Java page (`http://www.mozilla.org/rhino/`) to learn about Rhino.

■ **Note** Mozilla Rhino version 1.7R3 is included with Java 7. This implementation supports most of ECMAScript 5 (Strict Mode isn't supported), has partial support for JavaScript 1.8 (including expression closures and the destructuring assignment shorthand, but not generator expressions), and other features.

Obtaining Script Engines from Factories

Before performing other scripting tasks, a Java application must obtain an appropriate script engine. A script engine exists as an instance of a class that implements the `ScriptEngine` interface or extends the `AbstractScriptEngine` class. The application begins this task by creating an instance of the `ScriptEngineManager` class via one of these constructors:

- The `ScriptEngineManager()` constructor works with the calling thread's context classloader when one is available, or the bootstrap classloader otherwise, and a discovery mechanism to locate `ScriptEngineFactory` providers. (I discuss classloaders in Appendix C.)

- The `ScriptEngineManager(ClassLoader loader)` constructor works with the specified classloader and the discovery mechanism to locate `ScriptEngineFactory` providers. Passing `null` to `loader` is equivalent to calling the former constructor.

■ **Note** The discovery mechanism is documented in Appendix C under the "Extension Mechanism and ServiceLoader API" topic.

The application next uses the `ScriptEngineManager` instance to obtain a list of factories via this class's `List<ScriptEngineFactory> getEngineFactories()` method. For each factory, `ScriptEngineFactory` methods, such as `String getEngineName()`, return metadata describing the factory's script engine. Listing B-1 presents an application that demonstrates most of the metadata methods.

Listing B-1. *Enumerating script engines*

```
import java.util.List;


import javax.script.ScriptEngineFactory;
import javax.script.ScriptEngineManager;


class EnumerateScriptEngines
{
   public static void main(String[] args)
   {
      ScriptEngineManager manager = new ScriptEngineManager();
      List<ScriptEngineFactory> factories = manager.getEngineFactories();
      for (ScriptEngineFactory factory: factories)
      {
         System.out.println("Engine name (full): "+factory.getEngineName());
         System.out.println("Engine version: "+factory.getEngineVersion());
         System.out.println("Supported extensions:");
         List<String> extensions = factory.getExtensions();
         for (String extension: extensions)
            System.out.println("  "+extension);
         System.out.println("Language name: "+
                            factory.getLanguageName());
         System.out.println("Language version: "+
```

```
                        factory.getLanguageVersion());
        System.out.println("Supported MIME types:");
        List<String> mimetypes = factory.getMimeTypes();
        for (String mimetype: mimetypes)
           System.out.println("  "+mimetype);
        System.out.println("Supported short names:");
        List<String> shortnames = factory.getNames();
        for (String shortname: shortnames)
           System.out.println("  "+shortname);
        System.out.println();
     }
  }
}
```

Assuming that no additional script engines have been installed, you should observe the following output when you run this application against Java 7:

```
Engine name (full): Mozilla Rhino
Engine version: 1.7 release 3 PRERELEASE
Supported extensions:
  js
Language name: ECMAScript
Language version: 1.8
Supported MIME types:
  application/javascript
  application/ecmascript
  text/javascript
  text/ecmascript
Supported short names:
  js
  rhino
  JavaScript
  javascript
  ECMAScript
  ecmascript
```

The output reveals that an engine can have both a full name (`Mozilla Rhino`) and multiple short names (`rhino`, for example). The short name is more useful than the full name, as you will

see. It also shows that an engine can be associated with multiple extensions and multiple MIME types, and that the engine is associated with a scripting language.

`ScriptEngineFactory`'s `getEngineName()` and a few other metadata methods defer to `ScriptEngineFactory`'s `Object getParameter(String key)` method, which returns the script-engine-specific value associated with the argument passed to `key`, or null when the argument isn't recognized.

Methods such as `getEngineName()` invoke `getParameter()` with `key` set to an appropriate `ScriptEngine` constant, such as `ScriptEngine.ENGINE`. As Listing B-2 demonstrates, you can also pass `"THREADING"` to `key`, to identify a script engine's threading behavior, which you need to know when you plan to evaluate multiple scripts concurrently. `getParameter()` returns null when the engine isn't thread safe, or one of `"MULTITHREADED"`, `"THREAD-ISOLATED"`, or `"STATELESS"` to identify specific threading behavior.

Listing B-2. *Threading behavior*

```
import java.util.List;


import javax.script.ScriptEngineFactory;

import javax.script.ScriptEngineManager;


class ThreadingBehavior

{

   public static void main(String[] args)

   {

      ScriptEngineManager manager = new ScriptEngineManager();

      List<ScriptEngineFactory> factories = manager.getEngineFactories();

      for (ScriptEngineFactory factory: factories)

         System.out.println("Threading behavior: "+

                            factory.getParameter("THREADING"));

   }

}
```

Assuming that Mozilla Rhino 1.7 Release 3 is the only installed script engine, `ThreadingBehavior` outputs `Threading behavior: MULTITHREADED`. Scripts can execute concurrently on different threads, although the effects of executing a script on one thread might be visible to threads executing on other threads. Check out the `getParameter()` section of `ScriptEngineFactory`'s Java documentation to learn more about threading behaviors.

After determining the appropriate script engine, the application can invoke `ScriptEngineFactory`'s `ScriptEngine getScriptEngine()` method to return an instance of the script engine associated with the factory. Although new script engines are usually returned, a factory implementation is free to pool, reuse, or share implementations. The following example shows you how to accomplish this task:

```
if (factory.getLanguageName().equals("ECMAScript"))

{
```

```
   engine = factory.getScriptEngine();
   break;
}
```

Think of the example as being part of Listing B-1's/B-2's `for (ScriptEngineFactory factory: factories)` loop; assume that the `ScriptEngine` variable `engine` already exists. When the scripting language hosted by the factory is `ECMAScript` (language version does not matter in this example), a script engine is obtained from the factory and the loop is terminated.

Because the previous approach to obtaining a script engine is cumbersome, `ScriptEngineManager` provides three convenience methods that take on this burden, listed in Table B-2. These methods let you obtain a script engine based on file extension (possibly obtained via a dialog-selected script file), MIME type (possibly returned from a server), and short name (possibly chosen from a menu).

Table B-2. *`ScriptEngineManager` Convenience Methods for Obtaining a Script Engine*

Class/Interface	Description
`ScriptEngine getEngineByExtension(String extension)`	Create and return a script engine that corresponds to the given extension. When a script engine isn't available, this method returns null. An instance of the `java.lang.NullPointerException` class is thrown when `null` is passed to `extension`.
`ScriptEngine getEngineByMimeType(String mimeType)`	Create and return a script engine that corresponds to the given MIME type. When a script engine isn't available, this method returns null. A `NullPointerException` object is thrown when `null` is passed to `mimeType`.
`ScriptEngine getEngineByName(String shortName)`	Create and return a script engine that corresponds to the given short name. When a script engine isn't available, this method returns null. A `NullPointerException` object is thrown when `null` is passed to `shortName`.

Listing B-3 presents an application that invokes `getEngineByExtension()`, `getEngineByMimeType()`, and `getEngineByName()` to obtain a Rhino script engine instance. Behind the scenes, these methods take care of enumerating factories and invoking `ScriptEngineFactory`'s `getScriptEngine()` method to create the script engine.

Listing B-3. *Obtaining a script engine*

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


class ObtainScriptEngine
{
```

```
   public static void main(String[] args)
   {
      ScriptEngineManager manager = new ScriptEngineManager();
      ScriptEngine engine1 = manager.getEngineByExtension("js");
      System.out.println(engine1);
      ScriptEngine engine2 =
         manager.getEngineByMimeType("application/javascript");
      System.out.println(engine2);
      ScriptEngine engine3 = manager.getEngineByName("rhino");
      System.out.println(engine3);
   }
}
```

After compiling `ObtainScriptEngine.java`, running the application generates output that's similar to the following, indicating that different script engine instances are returned:

```
com.sun.script.javascript.RhinoScriptEngine@1fbc355
com.sun.script.javascript.RhinoScriptEngine@1d532ae
com.sun.script.javascript.RhinoScriptEngine@1f2428d
```

After a script engine has been obtained (via `ScriptEngineFactory`'s `getScriptEngine()` method or one of `ScriptEngineManager`'s three convenience methods), an application can access the engine's factory via `ScriptEngine`'s convenient `ScriptEngineFactory getFactory()` method. The application can also invoke various `ScriptEngine` methods to evaluate scripts.

■ **Note** `ScriptEngineManager` provides `void registerEngineExtension(String extension, ScriptEngineFactory factory)`, `void registerEngineMimeType(String type, ScriptEngineFactory factory)`, and `void registerEngineName(String name, ScriptEngineFactory factory)` methods that let applications dynamically register script engine factories with the script engine manager. Because these methods circumvent the discovery mechanism, you can replace an existing script engine factory and script engine with your own implementation, which is returned in subsequent calls to the "`getEngine`" methods.

Evaluating Scripts

After obtaining a script engine, an application can work with `ScriptEngine`'s six overloaded `eval()` methods to evaluate scripts. Each method throws a `ScriptException` instance when there's a problem with the script. Assuming successful script evaluation, an `eval()` method returns the script's result as some kind of `java.util.Object` instance, or null when the script doesn't return a value.

The simplest of the `eval()` methods are `Object eval(String script)` and `Object eval(Reader reader)`. The former method is invoked to evaluate a script expressed as a `String`;

the latter method is invoked to read a script from some other source (e.g., a file) and evaluate the script. Each method throws a `NullPointerException` instance when its argument is `null`. Listing B-4 demonstrates these methods.

Listing B-4. *Evaluating a Rhino-based script's functions*

```
import java.io.FileReader;
import java.io.IOException;

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

class FuncEvaluator
{
   public static void main(String[] args)
   {
      if (args.length != 2)
      {
         System.err.println("usage: java FuncEvaluator scriptfile "+
                            "script-exp");
         return;
      }
      ScriptEngineManager manager = new ScriptEngineManager();
      ScriptEngine engine = manager.getEngineByName("rhino");
      try
      {
         System.out.println(engine.eval(new FileReader(args[0])));
         System.out.println(engine.eval(args[1]));
      }
      catch (ScriptException se)
      {
         System.err.println(se.getMessage());
      }
      catch (IOException ioe)
      {
         System.err.println(ioe.getMessage());
```

```
      }
   }
}
```

`FuncEvaluator` is designed to evaluate the functions in a Rhino-based script file via `eval(Reader reader)`. It also uses `eval(String script)` to evaluate an expression that invokes one of the functions. Both the script file and script expression are passed to `FuncEvaluator` as command-line arguments. Listing B-5 presents a sample script file.

Listing B-5. *A statistics script (in `stats.js`) presenting `combinations()` and `fact()` JavaScript functions*

```
function combinations(n, r)
{
   return fact(n)/(fact(r)*fact(n-r))
}
function fact(n)
{
   if (n == 0)
      return 1;
   else
      return n*fact(n-1);
}
```

The `stats.js` file presents `combinations(n, r)` and `fact(n)` functions as part of a statistics package. The `combinations(n, r)` function works with the factorial function to calculate and return the number of different combinations of `n` items taken `r` items at a time. For example, how many different poker hands in five-card draw poker (where five cards are dealt to each player) can be dealt from a full card deck?

Invoke `java FuncEvaluator stats.js combinations(52,5)` to discover the answer. After outputting a message such as `sun.org.mozilla.javascript.internal.InterpretedFunction@169c398` on the first line (to indicate that `stats.js` doesn't return a value), `FuncEvaluator` outputs `2598960.0` on the line below. The value returned from `combinations(52,5)` indicates that there are 2,598,960 possible poker hands.

■ **Note** Wikipedia's "Combination" entry (`http://en.wikipedia.org/wiki/Combination`) introduces the statistical concept of combinations. Also, Wikipedia's "Five-card draw" entry (`http://en.wikipedia.org/wiki/Five-card_draw`) introduces the five-card draw poker variation.

Interacting with Java Classes and Interfaces from Scripts

The Scripting API is associated with *Java language bindings*, which are mechanisms that let scripts access Java classes and interfaces, create objects, and invoke methods according to the syntax of the scripting language. To access a Java class or interface, this type must be prefixed with its fully qualified package name. For example, in a Rhino-based script, you would specify `java.lang.Math.PI` to access the `PI` member in Java's `Math` class. In contrast, specifying `Math.PI` accesses the `PI` member in JavaScript's `Math` object.

To avoid needing to specify package names throughout a Rhino-based script, the script can employ the `importPackage()` and `importClass()` built-in functions to import an entire package of Java types or only a single type, respectively. For example, `importPackage(java.awt);` imports all of package `java.awt`'s types, and `importClass(java.awt.Frame);` imports only the `Frame` type from this package.

■ **Note** According to the *Java Scripting Programmer's Guide* (`http://java.sun.com/javase/7/docs/technotes/guides/scripting/programmer_guide/index.html`), `java.lang` is not imported by default, to prevent conflicts with same-named JavaScript types—`Object`, `Math`, `Boolean`, and so on.

The problem with `importPackage()` and `importClass()` is that they pollute JavaScript's global variable scope. Rhino overcomes this problem by providing `JavaImporter`, which works with JavaScript's `with` statement to let you specify classes and interfaces without their package names from within this statement's scope. Listing B-6's `swinggui.js` script demonstrates `JavaImporter`.

Listing B-6. *A script that creates a Swing graphical user interface (GUI) consisting of a label on the event-dispatch thread*

```
function creategui()
{
   var swinggui = new JavaImporter(java.awt, javax.swing);
   with (swinggui)
   {
      println("Event-dispatching thread: "+EventQueue.isDispatchThread());
      var r = new java.lang.Runnable()
              {
                 run: function()
                 {
                    println("Event-dispatching thread: "+
                            EventQueue.isDispatchThread());
                    var frame = new JFrame("Swing GUI");
                    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
                    var label = new JLabel("Hello from JavaScript",
                                           JLabel.CENTER);
                    label.setPreferredSize(new Dimension(300, 200));
                    frame.add(label);
                    frame.pack();
                    frame.setVisible(true);
                }
            };
        EventQueue.invokeLater(r);
    }
}
```

This script (which can be evaluated via `java FuncEvaluator swinggui.js creategui()`) creates a Swing GUI (consisting of a label) on the event-dispatch thread. `JavaImporter` imports types from the `java.awt` and `javax.swing` packages, which are accessible from the `with` statement's scope. Because `JavaImporter` doesn't import `java.lang`'s types, `java.lang` must be prepended to `Runnable`.

■ **Note** Listing B-6 also demonstrates implementing Java's `java.lang.Runnable` interface in JavaScript via a syntax similar to Java's anonymous class syntax. You can learn more about this and other Java-interaction features (e.g., creating and using Java arrays from JavaScript) from the *Java Scripting Programmer's Guide*.

Communicating with Scripts via Script Variables

Previously, you learned that `eval()` can return a script's result as an object. Additionally, the Scripting API lets applications pass objects to scripts via *script variables*, and obtain script variable values as objects. `ScriptEngine` provides `void put(String key, Object value)` and `Object get(String key)` methods for these tasks. Both methods throw `NullPointerException` when `key` is `null`, `java.lang.IllegalArgumentException` when `key` is the empty string, and (according to the `SimpleBindings.java` source code) `java.lang.ClassCastException` when `key` isn't a `String`. Listing B-7's application demonstrates `put()` and `get()`.

Listing B-7. *Using script variables to pass information to a script*

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

class MonthlyPayment
{
```

```
   public static void main(String[] args)
   {
      ScriptEngineManager manager = new ScriptEngineManager();
      ScriptEngine engine = manager.getEngineByExtension("js");
      // Script variables intrate, principal, and months must be defined (via
      // the put() method) prior to evaluating this script.
      String calcMonthlyPaymentScript =
         "intrate = intrate/1200.0;"+
         "payment = principal*intrate*(Math.pow(1+intrate, months)/"+
         "                             (Math.pow(1+intrate,months)-1));";
      try
      {
         engine.put("principal", 20000.0);
         System.out.println("Principal = "+engine.get("principal"));
         engine.put("intrate", 6.0);
         System.out.println("Interest Rate = "+engine.get("intrate")+"%");
         engine.put("months", 360);
         System.out.println("Months = "+engine.get("months"));
         engine.eval(calcMonthlyPaymentScript);
         System.out.printf("Monthly Payment = %.2f\n",
                           engine.get("payment"));
      }
      catch (ScriptException se)
      {
         System.err.println(se.getMessage());
      }
   }
}
```

`MonthlyPayment` calculates the monthly payment on a loan via the formula $MP = P*I*(1+I)^N/(1+I)^N\text{-}1$, where MP is the monthly payment, P is the principal, I is the interest rate divided by 1200, and N is the number of monthly periods to amortize the loan. Running this application with P set to 20000, I set to 6%, and N set to 360 results in this output:

```
Principal = 20000.0
Interest Rate = 6.0%
Months = 360
Monthly Payment = 119.91
```

The script depends on the existence of script variables `principal`, `intrate`, and `months`. These variables (with their object values) are introduced to the script via the `put()` method – `20000.0` and `6.0` are boxed into `java.lang.Double`s; `360` is boxed into a `java.lang.Integer`. The calculation result is stored in the `payment` script variable. `get()` returns this `Double`'s value to Java. (The `get()` method returns null when `key` doesn't exist.)

Applications are free to choose any syntactically correct string-based key (based on scripting language syntax) for a script variable's name, except for those keys beginning with the `javax.script` prefix. The Scripting API reserves this prefix for special purposes. Table B-3 lists several keys that begin with this prefix, together with their `ScriptEngine` constants.

Table B-3. *Reserved Keys and Their Constants*

Key	Constant	Description
`javax.script.argv`	`ARGV`	An `Object[]` array of arguments.
`javax.script.engine`	`ENGINE`	The full name of the script engine.
`javax.script.engine_version`	`ENGINE_VERSION`	The script engine's version.
`javax.script.filename`	`FILENAME`	The name of the script file being evaluated.
`javax.script.language`	`LANGUAGE`	The name of the scripting language associated with the script engine.
`javax.script.language_version`	`LANGUAGE_VERSION`	The version of the scripting language associated with the script engine.
`Javax.script.name`	`NAME`	The short name of the script engine.

Apart from `ARGV` and `FILENAME`, `ScriptEngineFactory` methods such as `getEngineName()` pass these constants as arguments to the previously discussed `getParameter(String key)` method. A Java application typically passes `ARGV` and `FILENAME` variables to a script, as in the following examples:

```
engine.put(ScriptEngine.ARGV, new String[] { "arg1", "arg2" });
engine.put(ScriptEngine.FILENAME, "file.js");
```

■ **Note** The `jrunscript` tool that Java provides to support scripting employs `engine.put("arguments", args)` followed by `engine.put(ScriptEngine.ARGV, args)` to make its command-line arguments available to a script. It also uses `engine.put(ScriptEngine.FILENAME, name)` to make the name of the script file being evaluated available to a script. The `jrunscript` tool is discussed under "Playing with the Command-Line Script Shell" later in this appendix.

Understanding Bindings and Scopes

The `put()` and `get()` methods interact with an internal map that stores key/value pairs. They access this map via an object whose class implements the `Bindings` interface, such as `SimpleBindings`. To determine which *bindings* objects are accessible to script engines, the Scripting API associates a *scope* identifier with each bindings object:

- The `ScriptContext.ENGINE_SCOPE` constant identifies the engine scope. A bindings object that's associated with this identifier is visible to a specific script engine throughout the engine's lifetime; other script engines don't have access to this bindings object, unless you share it with them. `ScriptEngine`'s `put()` and `get()` methods always interact with bindings objects that are engine scoped.
- The `ScriptContext.GLOBAL_SCOPE` constant identifies the global scope. A bindings object that's associated with this identifier is visible to all script engines that are created with the same script engine manager. `ScriptEngineManager`'s `void put(String key, Object value)` and `Object get(String key)` methods always interact with bindings objects that are globally scoped.

A script engine's bindings object for either scope can be obtained via `ScriptEngine`'s `Bindings getBindings(int scope)` method, with `scope` set to the appropriate constant. This object can be replaced via the `void setBindings(Bindings bindings, int scope)` method. `ScriptEngineManager`'s `Bindings getBindings()` and `void setBindings(Bindings bindings)` methods obtain/replace global bindings.

■ **Note** To share the global scope's bindings object with a newly created script engine, `ScriptEngineManager`'s `getEngineByExtension()`, `getEngineByMimeType()`, and `getEngineByName()` methods invoke `ScriptEngine`'s `setBindings()` method with `scope` set to `ScriptContext.GLOBAL_SCOPE`.

An application can create an empty `Bindings` object via `ScriptEngine`'s `Bindings createBindings()` method, and can temporarily replace a script engine's current bindings object with this new bindings object via `ScriptEngine`'s `getBindings()` and `setBindings()` methods. However, it's easier to pass this object to the `Object eval(String script, Bindings n)` and `Object eval(Reader reader, Bindings n)` methods, which also leave the current bindings unaffected. Listing B-8 presents an application that follows this approach and demonstrates various binding-oriented methods.

Listing B-8. *Getting to know bindings and scopes*

```
import javax.script.Bindings;

import javax.script.ScriptContext;

import javax.script.ScriptEngine;

import javax.script.ScriptEngineManager;

import javax.script.ScriptException;
```

```
public class GetToKnowBindingsAndScopes
{
   public static void main(String[] args)
   {
      ScriptEngineManager manager = new ScriptEngineManager();
      manager.put("global", "global bindings");
      System.out.println("INITIAL GLOBAL SCOPE BINDINGS");
      dumpBindings(manager.getBindings());
      ScriptEngine engine = manager.getEngineByExtension("js");
      engine.put("engine", "engine bindings");
      System.out.println("ENGINE'S GLOBAL SCOPE BINDINGS");
      dumpBindings(engine.getBindings(ScriptContext.GLOBAL_SCOPE));
      System.out.println("ENGINE'S ENGINE SCOPE BINDINGS");
      dumpBindings(engine.getBindings(ScriptContext.ENGINE_SCOPE));
      try
      {
         Bindings bindings = engine.createBindings();
         bindings.put("engine", "overridden engine bindings");
         bindings.put("app", new GetToKnowBindingsAndScopes());
         bindings.put("bindings", bindings);
         System.out.println("ENGINE'S OVERRIDDEN ENGINE SCOPE BINDINGS");
         engine.eval("app.dumpBindings(bindings);", bindings);
      }
      catch (ScriptException se)
      {
         System.err.println(se.getMessage());
      }
      ScriptEngine engine2 = manager.getEngineByExtension("js");
      engine2.put("engine2", "engine2 bindings");
      System.out.println("ENGINE2'S GLOBAL SCOPE BINDINGS");
      dumpBindings(engine2.getBindings(ScriptContext.GLOBAL_SCOPE));
      System.out.println("ENGINE2'S ENGINE SCOPE BINDINGS");
      dumpBindings(engine2.getBindings(ScriptContext.ENGINE_SCOPE));
      System.out.println("ENGINE'S ENGINE SCOPE BINDINGS");
```

```
      dumpBindings(engine.getBindings(ScriptContext.ENGINE_SCOPE));
   }
   public static void dumpBindings(Bindings bindings)
   {
      if (bindings == null)
         System.out.println("  No bindings");
      else
         for (String key: bindings.keySet())
            System.out.println("  "+key+": "+bindings.get(key));
      System.out.println();
   }
}
```

Because the global bindings are initially empty, the application adds a single `global` entry to these bindings. It then creates a script engine and adds a single `engine` entry to the script engine's initial engine bindings. Next, an empty bindings object is created and populated with a new `engine` entry via the `Bindings` interface's `Object put(String name, Object value)` method. New `app` and `bindings` entries are also added so that the script can invoke the application's `dumpBindings(Bindings bindings)` method to reveal the passed `Bindings` object's entries – `dumpBindings()` and the class in which this method is declared must be declared `public`. Finally, a second script engine is created, and an `engine` entry (with a value that differs from the first script engine's `engine` entry) is added to its default engine bindings. These tasks lead to output that's similar to the following:

```
INITIAL GLOBAL SCOPE BINDINGS
  global: global bindings


ENGINE'S GLOBAL SCOPE BINDINGS
  global: global bindings


ENGINE'S ENGINE SCOPE BINDINGS
  engine: engine bindings


ENGINE'S OVERRIDDEN ENGINE SCOPE BINDINGS
  app: GetToKnowBindingsAndScopes@1603bdc
  println: sun.org.mozilla.javascript.internal.InterpretedFunction@fa21a4
  engine: overridden engine bindings
  bindings: javax.script.SimpleBindings@1b6a1c4
  context: javax.script.SimpleScriptContext@1368c5d
  print: sun.org.mozilla.javascript.internal.InterpretedFunction@3945e2
```

```
ENGINE2'S GLOBAL SCOPE BINDINGS
  global: global bindings

ENGINE2'S ENGINE SCOPE BINDINGS
  engine2: engine2 bindings

ENGINE'S ENGINE SCOPE BINDINGS
  engine: engine bindings
```

The output shows that all script engines access the same global bindings, and that each engine has its own private engine bindings. It also reveals that passing a bindings object to a script via an `eval()` method doesn't affect the script's engine's current engine bindings. Finally, the output shows three interesting script variables – `println`, `print`, and `context` – which I discuss shortly.

■ **Tip** The `Bindings` interface presents a `void putAll(Map<? extends String, ? extends Object> toMerge)` method that's convenient for merging the contents of one bindings object with another bindings object.

Understanding Script Contexts

`ScriptEngine`'s `getBindings()` and `setBindings()` methods ultimately defer to `ScriptContext`'s equivalent methods of the same name. `ScriptContext` describes a *script context*, which connects a script engine to an application. It exposes the global and engine bindings objects, as well as a `java.io.Reader` instance and a pair of `java.io.Writer` instances that a script engine uses for input and output.

Every script engine has a default script context, which a script engine's constructor creates as an instance of `SimpleScriptContext`. The default script context is set as follows:

- The engine scope's set of bindings is initially empty.
- There is no global scope.
- A `java.io.InputStreamReader` that receives input from `java.lang.System.in` is created as the reader.
- `java.io.PrintWriter`s that send output to `System.out` and `System.err` are created as writers.

■ **Note** After each of `ScriptEngineManager`'s three "`getEngine`" methods obtains a script engine from the engine's factory, the method stores a reference to the shared global scope in the engine's default script context.

The default script context can be accessed via `ScriptEngine`'s `ScriptContext getContext()` method, and replaced via the companion `void setContext(ScriptContext context)` method. The `eval(String script)` and `eval(Reader reader)` methods invoke `Object eval(String script, ScriptContext context)` and `Object eval(Reader reader, ScriptContext context)` with the default script context as the argument.

In contrast, the `eval(String script, Bindings n)` and `eval(Reader reader, Bindings n)` methods first create a new temporary script context with engine bindings set to `n`, and with global bindings set to the default context's global bindings. These methods then invoke `eval(String script, ScriptContext context)` and `eval(Reader reader, ScriptContext context)` with the new script context as the argument.

Although you can create your own script context and pass it to `eval(String script, ScriptContext context)` or `eval(Reader reader, ScriptContext context)`, you might choose to manipulate the default script context instead. For example, if you want to send a script's output to a GUI's text component, you might install a new writer into the default script context, as demonstrated in Listing B-9.

Listing B-9. *Redirecting script output to a GUI*

```
import java.awt.BorderLayout;
import java.awt.EventQueue;
import java.awt.GridLayout;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import java.io.PrintWriter;
import java.io.Writer;

import javax.script.Bindings;
import javax.script.ScriptContext;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

import javax.swing.JButton;
import javax.swing.JFrame;
```

```
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

class RedirectScriptOutputToGUI extends JFrame
{
   static ScriptEngine engine;
   public RedirectScriptOutputToGUI()
   {
      super("Redirect Script Output to GUI");
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      getContentPane().add(createGUI());
      pack();
      setVisible(true);
   }
   JPanel createGUI()
   {
      JPanel pnlGUI = new JPanel();
      pnlGUI.setLayout(new BorderLayout());
      JPanel pnl = new JPanel();
      pnl.setLayout(new GridLayout(2, 1));
      final JTextArea txtScriptInput = new JTextArea(10, 60);
      pnl.add(new JScrollPane(txtScriptInput));
      final JTextArea txtScriptOutput = new JTextArea(10, 60);
      pnl.add(new JScrollPane(txtScriptOutput));
      pnlGUI.add(pnl, BorderLayout.NORTH);
      GUIWriter writer = new GUIWriter(txtScriptOutput);
      PrintWriter pw = new PrintWriter(writer, true);
      engine.getContext().setWriter(pw);
      engine.getContext().setErrorWriter(pw);
      pnl = new JPanel();
      JButton btnEvaluate = new JButton("Evaluate");
      ActionListener actionEvaluate;
      actionEvaluate = new ActionListener()
```

```
                    {
                        public void actionPerformed(ActionEvent ae)
                        {
                            try
                            {
                                engine.eval(txtScriptInput.getText());
                                dumpBindings();
                            }
                            catch (ScriptException se)
                            {
                                JFrame parent;
                                parent = RedirectScriptOutputToGUI.this;
                                JOptionPane.
                                    showMessageDialog(parent,
                                                      se.getMessage());
                            }
                        }
                    };
        btnEvaluate.addActionListener(actionEvaluate);
        pnl.add(btnEvaluate);
        JButton btnClear = new JButton("Clear");
        ActionListener actionClear;
        actionClear = new ActionListener()
                    {
                        public void actionPerformed(ActionEvent ae)
                        {
                            txtScriptInput.setText("");
                            txtScriptOutput.setText("");
                        }
                    };
        btnClear.addActionListener(actionClear);
        pnl.add(btnClear);
        pnlGUI.add(pnl, BorderLayout.SOUTH);
        return pnlGUI;
    }
```

```
   static void dumpBindings()
   {
      System.out.println("ENGINE BINDINGS");
      Bindings bindings = engine.getBindings(ScriptContext.ENGINE_SCOPE);
      if (bindings == null)
         System.out.println("  No bindings");
      else
         for (String key: bindings.keySet())
            System.out.println("  "+key+": "+bindings.get(key));
      System.out.println();
   }
   public static void main(String[] args)
   {
      ScriptEngineManager manager = new ScriptEngineManager();
      engine = manager.getEngineByName("rhino");
      dumpBindings();
      Runnable r = new Runnable()
                   {
                      public void run()
                      {
                         new RedirectScriptOutputToGUI();
                      }
                   };
      EventQueue.invokeLater(r);
   }
}
class GUIWriter extends Writer
{
   private JTextArea txtOutput;
   GUIWriter(JTextArea txtOutput)
   {
      this.txtOutput = txtOutput;
   }
   public void close()
   {
```

```
        System.out.println("close");
    }
    public void flush()
    {
        System.out.println("flush");
    }
    public void write(char[] cbuf, int off, int len)
    {
        txtOutput.setText(txtOutput.getText()+new String(cbuf, off, len));
    }
}
```

`RedirectScriptOutputToGUI` creates a Swing GUI with two text components and two buttons. After entering a Rhino-based script into the upper text component, click the Evaluate button to evaluate the script. When there's a problem with the script, a dialog appears with an error message. Otherwise, the script's output appears in the lower text component. Click the Clear button to erase the contents of both text components. Figure B-1 shows the GUI.

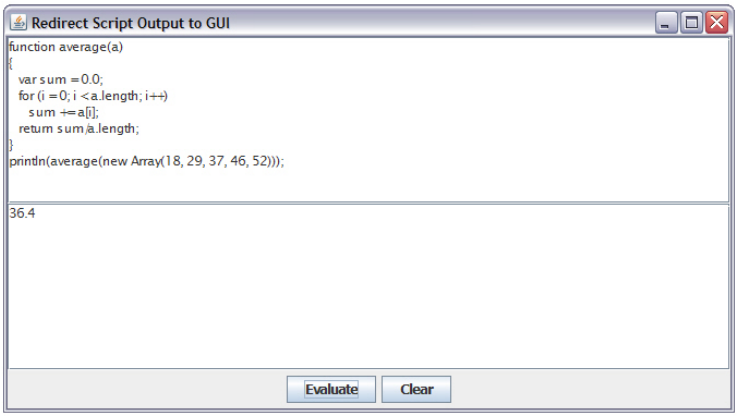

Figure B-1. *By installing a new writer into the default script context, you can send a script's output to a GUI's text component.*

To redirect a script's output to the lower text component, `RedirectScriptOutputToGUI` creates an instance of `GUIWriter` and makes this instance available to the script engine via `ScriptContext`'s `void setWriter(Writer writer)` and `void setErrorWriter(Writer writer)` methods. Although they're not used in the example, `ScriptWriter` also provides companion `Writer getWriter()` and `Writer getErrorWriter()` methods.

■ **Note** `ScriptContext` also provides a `void setReader(Reader reader)` method for changing a script's input source, and a `Reader getReader()` method for identifying the current input source.

As well as displaying script output in the GUI, `RedirectScriptOutputToGUI` also outputs the engine scope's bindings to the console window when you start this application, and each time you click Evaluate. Initially, there are no bindings. However, after clicking Evaluate, you'll discover `context`, `print`, and `println` script variables in the engine bindings.

The `context` script variable describes a `SimpleScriptContext` object that lets a script engine access the script context. The Rhino script engine needs to access the script context in order to implement the `print()` and `println()` functions. When you evaluate the `println(println);` script followed by the `println(print);` script, you'll discover output similar to the following:

```
function println(str) {
    print(str, true);
}
function print(str, newline) {
    if (typeof(str) == "undefined") {
        str = "undefined";
    } else {
        if (str == null) {
            str = "null";
        }
    }
    var out = context.getWriter();
    if (!(out instanceof java.io.PrintWriter)) {
        out = new java.io.PrintWriter(out);
    }
    out.print(String(str));
    if (newline) {
        out.print("\n");
    }
    out.flush();
}
```

The output reveals that the `context` script variable is needed to access the current writer, which happens to be the `GUIWriter` instance created in the `RedirectScriptOutputToGUI` application. This script variable also can be used to access arguments or the script's filename. For example, if this application invoked `engine.put(ScriptEngine.ARGV, new String[] {"A", "B", "C"});` followed by `engine.put(ScriptEngine.FILENAME, "script.js");` on a script engine referenced by the `ScriptEngine` variable `engine`, and you evaluated this script from the application's GUI via `println(context.getAttribute("javax.script.filename"));` `println(context.getAttribute("javax.script.argv")[0]);`, you would see `script.js` followed by `A` appear on separate lines in the lower text component.

Depending on your application, you might not want to "pollute" the default script context with new writers, bindings, and so on. Instead, you might want the same script to work in different contexts, leaving the default context untouched. To accomplish this task, create a

`SimpleScriptContext` instance, populate its engine bindings via `ScriptContext`'s `void setAttribute(String name, Object value, int scope)` method, and invoke `eval(String script, ScriptContext context)` or `eval(Reader reader, ScriptContext context)` with this script context. The following example allows the `script`-referenced script to access the engine bindings object `app` in a new context:

```
ScriptContext context = new ScriptContext();
context.setAttribute("app", this, ScriptContext.ENGINE_SCOPE);
Object result = engine.eval(script, context);
```

TIPS FOR WORKING WITH SCRIPT SCOPES AND CONTEXTS

The `setAttribute()` method is a convenient alternative to first accessing a scope's `Bindings` object and then invoking its `put()` method. For example, `context.setAttribute("app", this, ScriptContext.ENGINE_SCOPE);` is easier to express than `context.getBindings(ScriptContent.ENGINE_SCOPE).put("app", this);`.

You'll also find `ScriptContext`'s `Object getAttribute(String name, int scope)` and `Object removeAttribute(String name, int scope)` methods to be more convenient than the alternatives.

Finally, you'll find the following to be useful in situations where there are more than engine and global scopes:

- `Object getAttribute(String name)` returns the named attribute from the lowest scope.
- `int getAttributesScope(String name)` returns the lowest scope in which an attribute is defined.
- `List<Integer> getScopes()` returns an immutable list of valid scopes for the script context.

It's possible to subclass `SimpleScriptContext` and define a new scope (perhaps for use by servlets) that coincides with this context. However, doing so is beyond this appendix's scope (no pun intended).

Generating Scripts from Macros

Many applications benefit from *macros* (named sequences of commands/instructions that automate various tasks). For example, Word and other Microsoft Office products include a macro language called Visual Basic for Applications (VBA), which lets users create macros to automate editing, formatting, and other tasks.

■ **Note** Check out Wikipedia's "Macro (computer science)" entry (`http://en.wikipedia.org/wiki/Macro_(computer_science)`) for a refresher on macros and macro languages.

An application that parses a macro generates an equivalent script in some scripting language. Because script syntax differs from one scripting language to another, it's important that the application be able to generate the script in a portable manner so that it can be easily adapted to various scripting languages. The `ScriptEngineFactory` interface declares three methods for this purpose, as listed in Table B-4.

Table B-4. *`ScriptEngineFactory` Methods for Generating Scripts from Macros*

Class/Interface	Description
`String getMethodCallSyntax(String obj, String m, String... args)`	Return a `String` that can be used to invoke a Java object's method using a scripting language's syntax. Parameter `obj` identifies the object whose method is to be invoked, parameter `m` is the name of the method to be invoked, and parameter `args` identifies the names of the method's arguments. For example, invoking `getMethodCallSyntax("x", "factorial", new String[] {"num"})` might return `"$x->factorial($num);"` for a PHP script engine. PHP variable names are prefixed with a dollar-sign character.
`String getOutputStatement(String toDisplay)`	Return a `String` that can be used as a statement to output the argument passed to `toDisplay` using the scripting language's syntax. For example, invoking `getOutputStatement("Hello")` might return `"echo(\"Hello\");"` for a PHP script engine, whereas it returns `"print(\"Hello\")"` for the Rhino JavaScript script engine.
`String getProgram(String... statements)`	Return a `String` that organizes the specified `statements` into a valid script using the scripting language's syntax. For example, assuming that variable `factory` references a `ScriptEngineFactory` instance, invoking `factory.getProgram(factory.getOutputStatement(factory.getMethodCallSyntax("x", "factorial", new String[] {"num"})));` might return `"<? echo($x->factorial($num)); ?>"` for a PHP script engine.

Do you notice a problem with `getOutputStatement()`? Although you might expect the Rhino script engine's `getOutputStatement()` method to be implemented as `return "print("+toDisplay+")";`, this method is implemented as `return "print(\""+toDisplay+"\")";`. In other words, anything passed in `toDisplay` is surrounded by double quotation marks. This is problematic when you want to pass a variable name to `getOutputStatement()`, and expect to obtain an output statement that outputs the variable's contents instead of its name.

You can easily solve this problem by replacing the double quotation marks with spaces. Assuming that `os` is a `String` variable holding `getOutputStatement()`'s result, `os = os.replace('"', ' ');` replaces the double quotation marks with spaces. Because this problem

might be addressed in a future version of the Rhino script engine, it's best to first verify the version number, as in: `if (factory.getEngineVersion().equals("1.7 release 3 PRERELEASE")) os = os.replace('"', ' ');`.

Compiling Scripts

Script engines tend to evaluate scripts via *interpreters*, which can be conceptualized as consisting of a *front end* for parsing source code and generating intermediate code, and a *back end* for executing the intermediate code. Every time a script is evaluated, the parsing and intermediate code-generation tasks are performed before execution, which tends to slow down script evaluation.

To hasten a script's evaluation, many script engines allow intermediate code to be stored and executed repeatedly. A script engine class that supports this *compilation* feature implements the optional `Compilable` interface. `Compilable`'s methods compile scripts into intermediate code and store results in `CompiledScript` subclass objects, whose `eval()` methods execute the intermediate code.

■ **Note** I refer to scripts as being *evaluated* instead of *executed*. After all, `ScriptEngine` specifies `eval()` methods, not `exec()` methods. In contrast, I refer to intermediate code as being *executed*, to be somewhat consistent with JSR 223.

An application must cast a script engine object to a `Compilable` before it can compile a script. Before doing this, the application should make sure that the engine's class implements the `Compilable` interface. Note that this isn't necessary for the Rhino script engine, which supports `Compilable`. The following code fragment (which assumes the existence of an `engine` object) demonstrates this task:

```
Compilable compilable = null;
if (engine instanceof Compilable)
   compilable = (Compilable) engine;
```

The `Compilable` interface presents `CompiledScript compile(String script)` and `CompiledScript compile(Reader script)` methods for compiling a script and returning its intermediate code via a `CompiledScript` subclass object. Both methods throw a `NullPointerException` instance when the argument is `null`, and a `ScriptException` instance when there's a problem with the script.

The `CompiledScript` class includes `Object eval()`, `Object eval(Bindings bindings)`, and `abstract Object eval(ScriptContext context)` methods for executing the script's intermediate code. Each method throws a `ScriptException` instance when a script error occurs at runtime. `CompiledScript` also includes an `abstract ScriptEngine getEngine()` method that provides access to the compiled script's engine.

What kind of speed improvement can you expect from compilation? To answer this question, I've created an application that presents a simple script consisting of a factorial function, evaluates this script 10,000 times, compiles the script, and executes the script's intermediate code 10,000 times. Each loop is timed to see how long it takes to run. The source code for this application appears in Listing B-10.

Listing B-10. *Testing compilation speed*

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

class TestCompilationSpeed
{
   final static int MAX_ITERATIONS = 10000;
   public static void main(String[] args) throws Exception
   {
      ScriptEngineManager manager = new ScriptEngineManager();
      ScriptEngine engine = manager.getEngineByName("JavaScript");
      String fact = "function fact(n)"+
                    "{"+
                    "   if (n == 0)"+
                    "      return 1;"+
                    "   else"+
                    "      return n*fact(n-1);"+
                    "};";
      long time = System.currentTimeMillis();
      for (int i = 0; i < MAX_ITERATIONS; i++)
         engine.eval(fact);
      System.out.println(System.currentTimeMillis()-time);
      Compilable compilable = null;
      if (engine instanceof Compilable)
      {
         compilable = (Compilable) engine;
         CompiledScript script = compilable.compile(fact);
         time = System.currentTimeMillis();
         for (int i = 0; i < MAX_ITERATIONS; i++)
            script.eval();
         System.out.println(System.currentTimeMillis()-time);
      }
   }
}
```

Each time you run this application, you'll probably notice slightly different results. However, these results show a significant speed improvement. For example, you might see that the evaluated script took 2375 milliseconds and the compiled script took 1328 milliseconds.

■ **Note** `TestCompilationSpeed` doesn't assume that `JavaScript` corresponds to the Mozilla Rhino 1.7 Release 3 script engine. Recall that a script engine factory and its script engine can be overridden by any of `ScriptEngineManager`'s "`registerEngine`" methods. For this reason, `TestCompilationSpeed` verifies that the engine's class implements `Compilable`, even though Rhino's script engine class implements this interface.

Invoking Global, Object Member, and Interface-Implementing Functions

In contrast to compilation, which allows the intermediate code of entire scripts to be reexecuted, the Scripting API's support for *invocation* allows the intermediate code of only global functions and object member functions to be reexecuted. Furthermore, these functions can be invoked directly from Java code, which can pass object arguments to and return object results from these functions.

A script engine class that supports invocation implements the optional `Invocable` interface. An application must cast a script engine object to an `Invocable` instance before it can invoke global functions and object member functions. As with `Compilable`, your application should first verify that a script engine supports `Invocable` before casting. And again, this isn't necessary for Rhino, which supports `Invocable`.

The `Invocable` interface provides an `Object invokeFunction(String name, Object... args)` method to invoke a global function. The global function's name is identified by `name`, and arguments to be passed to the global function are identified by `args`. If the global function is successful, this method returns its result as an `Object`. Otherwise, the method throws `ScriptException` when something goes wrong during the global function's invocation, `java.lang.NoSuchMethodException` when the global function cannot be found, and `NullPointerException` when a null reference is passed to `name`. The following example (which assumes the existence of a Rhino-based `engine` object) demonstrates `invokeFunction()`:

```
// The script presents a global function that converts degrees
// Celsius to degrees Fahrenheit.
String script = "function c2f(degrees)"+
                "{"+
                "   return degrees*9.0/5.0+32;"+
                "}";
// First evaluate the script, to generate intermediate code.
engine.eval(script);
// Then invoke the script's c2f() global function with an argument
// that will be boxed into a Double. After passing the argument to
// the global function, its intermediate code will be executed, and
```

```
// a value will be returned to Java as a Double containing 212.0.
Invocable invocable = (Invocable) engine;
System.out.println(invocable.invokeFunction("c2f", 200.0));
```

The `Invocable` interface provides an `Object invokeMethod(Object thiz, String name, Object... args)` method to invoke an object member function. The script object's reference (obtained after a previous script evaluation or via a prior invocation) is identified by `thiz`, the member function's name is identified by `name`, and arguments passed to the member function are identified by `args`. Upon success, the member function's result is returned as an `Object`. As well as `invokeFunction()`'s exceptions, `IllegalArgumentException` is thrown when either `null` or an `Object` reference not representing a script object is passed to `thiz`. The following code fragment demonstrates `invokeMethod()`:

```
// The script presents an object with a member function that
// converts degrees Celsius to degrees Fahrenheit.
String script = "var obj = new Object();"+
                "obj.c2f = function(degrees)"+
                "{"+
                "   return degrees*9.0/5.0+32;"+
                "}";
// First evaluate the script, to generate intermediate code.
engine.eval(script);
// Then get script object whose member function is to be invoked.
Object obj = engine.get("obj");
// Finally, invoke the c2f() member function with an argument that
// will be boxed into a Double. After passing the argument to the
// global function, its intermediate code will be executed, and
// a value will be returned to Java as a Double containing 98.6.
Invocable invocable = (Invocable) engine;
System.out.println(invocable.invokeMethod(obj, "c2f", 37.0));
```

Directly invoking a script's global and object member functions results in an application being strongly coupled to the script. As changes are made to function names and their parameter lists, the application must adapt. To minimize coupling, `Invocable` provides two methods that return Java interface objects, whose methods are implemented by a script's global and object member functions:

- `<T> T getInterface(Class<T> clazz)` returns an implementation of the `clazz`-identified interface, where the methods are implemented by a script's global functions.
- `<T> T getInterface(Object thiz, Class<T> clazz)` returns an implementation of the `clazz`-identified interface, where the methods are implemented by scripting object `thiz`'s member functions.

Both methods return null when the requested interface is unavailable, because the intermediate code is missing one or more functions that implement interface methods. `IllegalArgumentException` is thrown when `null` is passed to `clazz`, when `clazz` doesn't represent an interface, or when either `null` or an `Object` reference not representing a script object is passed to `thiz`.

The previous Java code fragments that demonstrated `invokeFunction()` and `invokeMethod()` worked directly with a `c2f()` global function. Because this function is coupled to the Java code, this code would need to be changed should `c2f()` be eliminated in favor of a more descriptive and generic global function, such as one that also converts to degrees Celsius. The generic global function's signature will not change, even when its implementation changes (you might start out calling `c2f()`, and later remove this function after integrating this other function's code into the implementation). As a result, the generic global function is a perfect choice for implementing a Java interface, as Listing B-11 demonstrates.

Listing B-11. *Implementing a Java interface*

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


class TemperatureConversion
{
   public static void main(String[] args) throws ScriptException
   {
      ScriptEngineManager manager = new ScriptEngineManager();
      ScriptEngine engine = manager.getEngineByName("rhino");
      String script = "function c2f(degrees)"+
                      "{"+
                      "   return degrees*9.0/5.0+32;"+
                      "}"+
                      " "+
                      "function f2c(degrees)"+
                      "{"+
                      "   return (degrees-32)*5.0/9.0;"+
                      "}"+
                      " "+
                      "function convertTemperature(degrees, toCelsius)"+
                      "{"+
                      "   if (toCelsius)"+
```

```
                    "      return f2c(degrees);"+
                    "   else"+
                    "      return c2f(degrees);"+
                    "}";
      engine.eval(script);
      Invocable invocable = (Invocable) engine;
      TempConversion tc = invocable.getInterface(TempConversion.class);
      if (tc == null)
          System.err.println("Unable to obtain TempConversion interface");
      else
      {
          System.out.println("37 degrees Celsius = "+
                             tc.convertTemperature(37.0, false)+
                             " degrees Fahrenheit");
          System.out.println("212 degrees Fahrenheit = "+
                             tc.convertTemperature(212.0, true)+
                             " degrees Celsius");
      }
   }
}
interface TempConversion
{
   double convertTemperature(double degrees, boolean toCelsius);
}
```

The application provides a `TempConversion` interface whose `double convertTemperature(double degrees, boolean toCelsius)` method corresponds to a same-named global function in the script. Executing `invocable.getInterface(TempConversion.class)` returns a `TempConversion` instance, which can be used to invoke `convertTemperature()`. Here's the application's output:

```
37 degrees Celsius = 98.6 degrees Fahrenheit
212 degrees Fahrenheit = 100.0 degrees Celsius
```

■ **Note** The *Java Scripting Programmer's Guide*'s "Implementing Java Interfaces by Scripts" section (`http://java.sun.com/javase/7/docs/technotes/guides/scripting/programmer_guide/index.htm`

l#interfaces) presents the source code for a pair of applications that further demonstrate the getInterface() methods.

Playing with the Command-Line Script Shell

Java provides jrunscript, an experimental command-line, script-shell tool for exploring scripting languages and their communication with Java. The JDK documentation's "jrunscript - command line script shell" page (http://java.sun.com/javase/7/docs/technotes/tools/share/jrunscript.html) offers a tool reference.

Although jrunscript can be used to evaluate file-based scripts or scripts that are specified on the command line, the easiest way to work with this tool is via interactive mode. In this mode, jrunscript prompts you to enter a line of code. It evaluates this code after you press the Enter key. To enter interactive mode, specify only jrunscript on the command line.

In response, you see the js> prompt. The js is a reminder that the default language is JavaScript (js is actually one of the short names for the Mozilla Rhino engine). At the js> prompt, you can enter Rhino JavaScript statements and expressions. When an expression is entered, its value will appear on the next line, as the following session demonstrates:

```
js> Math.PI // Access the PI member of JavaScript's Math object.

3.141592653589793
```

During its initialization, jrunscript introduces several built-in global functions to the Rhino script engine. These functions range from outputting the date in the current locale, to listing the files in the current directory and performing other filesystem tasks, to working with XML. The following session demonstrates a few of these functions:

```
js> date()

July 22, 2011 1:36:00 PM CDT

js> ls()

-rw Jul 22    950 swinggui.js

js> cat("swinggui.js", "frame")

13            :               var frame = new JFrame("Swing GUI");

14            :               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

18            :               frame.getContentPane().add(label);

19            :               frame.pack();

20            :               frame.setVisible(true);

js> load("swinggui.js")

js> creategui()

Event-dispatching thread: false

js> Event-dispatching thread: true
```

The example uses four jrunscript built-in functions:

- date() outputs the current date.

- `ls()` lists the current directory's files.
- `cat()` outputs part (based on a pattern match) or all of a file's contents.
- `load()` loads and evaluates a script file, such as `swinggui.js`.

After `creategui()` finishes, the Swing application's console output mixes with `jrunscript`'s console output. Closing the GUI also closes `jrunscript`.

As well as its utility functions, `jrunscript` provides `jlist()` and `jmap()` functions. `jlist()` lets you access a `java.util.List` instance like an array with integer indexes. `jmap()` is for accessing a `Map` instance like a Perl-style associative array with string-based keys. A `List` or `Map` instance is passed to `jlist()` or `jmap()` as an argument. The functions return an object that provides the access, as the following session demonstrates:

```
js> var scriptlanguages = new java.util.ArrayList()
js> scriptlanguages.add('JavaScript')
true
js> scriptlanguages.add('Ruby')
true
js> scriptlanguages.add('Groovy')
true
js> var sl = jlist(scriptlanguages)
js> sl[1]
Ruby
js> sl.length
3
js> println(sl)
[JavaScript, Ruby, Groovy]
js> delete sl[1]
false
js> println(sl)
[JavaScript, Groovy]
js> sl.length
2
js> var properties = java.lang.System.getProperties()
js> var props = jmap(properties)
js> props['java.version']
1.7.0
js> props['os.name']
Windows XP
js> delete props['os.name']
```

```
true
js> props['os.name']
js>
```

The session shows that `ArrayList` and `System` are prefixed with their `java.util` and `java.lang` package names, respectively. `jrunscript` doesn't import the `java.util` and `java.lang` packages by default, although it does import the `java.io` and `java.net` packages by default. This session also demonstrates the use of JavaScript's `delete` operator to delete list and map entries.

■ **Note** If you're wondering why `delete sl[1]` outputs `false`, whereas `delete props['os.name']` outputs `true`, the reason has to do with JavaScript's `delete` operator returning true only when the entry being deleted no longer exists. `Ruby` is removed from `sl[1]` by replacing `sl[1]`'s contents with `Groovy`. This implies that `sl[2]`, which previously contained `Groovy`, no longer exists. Although `delete sl[1]` accomplished the objective of removing `Ruby`, `sl[1]` still exists and contains `Groovy`. Hence, `delete sl[1]` outputs `false`. If `delete sl[2]` had been specified, `true` would have been output because `sl[2]` would no longer exist – there's no `sl[3]` with a value to shift into `sl[2]`.

The `jrunscript` tool introduces another built-in function to the Rhino script engine: `JSInvoker()`. As with `jlist()` and `jmap()`, this function returns a proxy object for a delegate object. `JSInvoker()`'s proxy is used to invoke the delegate's special `invoke()` member function via arbitrary member function names and argument lists. The following session provides a demonstration:

```
js> var x = { invoke: function(name, args) { println(name+" "+args.length); }};
js> var y = new JSInvoker(x);
js> y.run("first", "second", "third");
run 3
js> y.doIt();
doIt 0
js> y.doIt(10);
doIt 1
js>
```

Delegate object `x` specifies a single member function named `invoke`. This function's arguments are a string-based name and an array of object arguments. The second line employs `JSInvoker()` to create a proxy object that's assigned to `y`. By using this proxy object, you can call the delegate object's `invoke()` member function via an arbitrary name and number of arguments. Behind the scenes, `y.run("first", "second", "third")` translates into `x.invoke('run', args)`, where `args` is an array containing `"first"`, `"second"`, and `"third"` as its three entries. Also, `y.doIt()` translates into `x.invoke('doIt', args)`, where `args` is an empty array. A similar translation is performed on `y.doIt(10);`.

If you were to print the contents of the `jlist()`, `jmap()`, and `JSInvoker()` functions via `println(jlist)`, `println(jmap)`, and `println(JSInvoker)`, you would observe that these functions

are implemented by `JSAdapter`, a `java.lang.reflect.Proxy` equivalent for JavaScript. `JSAdapter` lets you adapt property access (as in `x.i`), mutator (as in `x.p = 10`), and other simple JavaScript syntax on a proxy object to a delegate JavaScript object's member functions.

To terminate `jrunscript` after playing with this tool, specify the `exit()` function with or without an exit-code argument. For example, you can specify `exit(0)`, or `exit()` by itself. When you specify `exit()` without an argument, 0 is chosen as the exit code. This code is returned from `jrunscript` for use in Windows batch files, Unix shell scripts, and so on. Alternatively, you can specify the `quit()` function, which is a synonym for `exit()`.

Dynamically Typed Language Support

Dynamically typed languages (e.g., JavaScript, Ruby, Python, Groovy, and SmallTalk) are widely used for several reasons. For one thing, their interpretive natures offer rapid turnaround times – companies such as RedHat and Twitter have increasingly focused on Ruby for this reason. Another reason is that programs created by dynamically typed languages can "change their shapes" (think of self-modifying code) as the data that they are monitoring changes. Such programs can evolve to handle more complex data-driven scenarios.

Various dynamically typed languages have been ported to the Java platform to take advantage of Java's benefits, such as its huge standard class library. For example, JavaScript/Rhino, JRuby, Jython, and Groovy run natively on the JVM.

Performance problems have arisen because languages targeting the JVM typically have their own ways of executing code (e.g., SmallTalk sends string-based messages that have to be looked up) whereas the JVM sees only method calls. Converting a dynamically typed language to run on the JVM often involves a certain amount of overhead to map its way of executing functional code to JVM method calls. This overhead impacts performance, especially because it often relies on the Reflection API. The Just-In-Time compiler (which I briefly introduced in Chapter 1) cannot perform various optimizations that would lead to better-performing code.

To address the performance dilemma, the Da Vinci Machine Project (see `http://openjdk.java.net/projects/mlvm/`) along with JSR 292 "Supporting Dynamically Typed Languages on the Java Platform" (`http://jcp.org/en/jsr/summary?id=292`) were launched a few years ago. The result is a revised JVM architecture that primarily consists of a new `invokedynamic` instruction. Furthermore, a Java API that largely consists of types in the `java.lang.invoke` package has been created. Key types within this package support *method handles*, which are analogous to function pointers in a language such as C, and which are used to connect *dynamic call sites* (instances of the `invokedynamic` instruction) to methods.

■ **Note** John Rose, Lead Engineer of the Da Vinci Machine Project and Specification Lead for JSR 292 discusses Java 7's support for dynamic languages in Oracle's "A Renaissance VM: One Platform, Many Languages" video at `http://medianetwork.oracle.com/media/show/16802`.

Rose does an excellent job of explaining the rationale for and architecture of Java 7's support for dynamically typed languages, and I encourage you to watch the video. Rather than reiterate his words, I'd like to focus on one tidbit: Toward the end of the video, Rose mentions that Oracle wants to use `invokedynamic` to implement Project Lambda for JDK 8 (expected to arrive in late 2012/early 2013).

According to the "Project Lambda" website's main page (`http://openjdk.java.net/projects/lambda/`) this project "aims to support programming in a multicore environment by adding closures and related features to the Java language." Wikipedia's

"Closure (computer science)" entry (`http://en.wikipedia.org/wiki/Closure_(computer_science)`) defines *closure* as "a function together with a referencing environment for the nonlocal variables of that function." Furthermore, it states that "a closure allows a function to access variables outside its typical scope. Such a function is 'closed over' its free variables. The referencing environment binds the nonlocal names to the corresponding variables in scope at the time the closure is created, additionally extending their lifetime to at least as long as the lifetime of the closure itself. When the closure is entered at a later time, possibly from a different scope, the function is executed with its nonlocal variables referring to the ones captured by the closure."

Consider the following Groovy closure example:

```
sayHello = { message -> println "hello "+message }

[ "davinci", "closure" ].each(sayHello)
```

This example introduces a closure named `sayHello`. (The presence of the { and } characters inform Groovy that this expression is to be treated as code – a closure.) Variable `message` is a parameter that takes on the argument passed to `sayHello()`. For example, `message` takes on `Java 7` when we specify `sayHello("Java 7")`, which results in the closure outputting `hello Java 7`.

Closures are very powerful because they can be passed as arguments to functions. For example, the second line invokes Groovy's `each()` function on each element in the list of two strings: `"davinci"` and `"closure"`. For each of these strings, `sayHello` is passed as an argument to `each()`, its `message` parameter is assigned one of the strings in this list, and its `println` function call outputs `hello`, a space, and the string value. The following output is generated:

```
hello davinci

hello closure
```

In his "method handles == closures ??" blog post (`http://weblogs.java.net/blog/2009/01/02/method-handles-closures`), developer Rémi Forax states that method handles are a good candidate for implementing closures. He also presents a Java 7 example that's the equivalent of the Groovy example. I've updated Forax's example to support the final release of JDK 7; Listing B-12 contains the complete example's source code.

Listing B-12. *Using a method handle to implement a closure*

```
import java.lang.invoke.MethodHandle;

import java.lang.invoke.MethodHandles;

import java.lang.invoke.MethodType;


import java.util.Arrays;

import java.util.Collection;

import java.util.List;


class ClosureDemo

{

   public static void each(Collection<?> c, MethodHandle mh) throws Throwable

   {

      for (Object element : c)
```

```
            mh.invoke(element);
        }
        public static void sayHello(Object message)
        {
            System.out.println("hello "+message);
        }
        public static void main(String[] args) throws Throwable
        {
            MethodHandles.Lookup LOOKUP = MethodHandles.lookup();
            MethodType methodType = MethodType.methodType(void.class, Object.class);
            MethodHandle mh = LOOKUP.findStatic(ClosureDemo.class, "sayHello",
                                                methodType);
            List list = Arrays.asList("davinci", "closure");
            each(list, mh);
        }
}
```

Listing B-12 describes an application for iterating over a `java.util.Collection` and calling the `sayHello()` class method on each element.

The `main()` method first executes `MethodHandles.lookup()` to return a *lookup object*, which is basically a factory for creating method handles.

`main()` next executes `MethodType`'s `MethodType methodType(Class<?> rtype, Class<?> ptype0)` class method to find or create a *method type*, which represents the arguments and return type accepted and returned by a method handle, or the arguments and return type passed and expected by a method handle caller. In this case, the method type specifies a return type placeholder of `void` and a parameter type of `Object` via class literals.

Continuing, `main()` uses the previously returned lookup object to invoke `MethodHandles.Lookup`'s `MethodHandle findStatic(Class<?> refc, String name, MethodType type)` method to obtain a method handle for the `sayHello()` class method.

After using `Arrays.asList()` to return a `List` implementation, `main()` calls the `each()` class method with this collection and the previously created method handle as arguments. For each element in the collection, `MethodHandle`'s `Object invoke(Object... args)` method is called to invoke the method handle, which results in the `sayHello()` method executing and outputting the argument passed to `invoke()`.

Compile Listing B-12 (`javac ClosureDemo.java`) and run the application (`java ClosureDemo`). You should observe the following output:

```
hello davinci
hello closure
```