Chapter 29: Introduction to Scripting in Java (e.g., JavaScript Engine)

29.0 Introduction

Traditionally, Java has been a statically typed and compiled programming language, but with the growing need for flexibility and dynamic behavior in applications, Java incorporated **scripting support** through the **Java Scripting API** (JSR 223). This allowed Java applications to **embed and execute scripts written in dynamic languages**, such as JavaScript, directly within Java code. This chapter explores Java's scripting capabilities, primarily using the **Nashorn JavaScript engine**, its API usage, and practical integration techniques.

29.1 What is Scripting in Java?

Scripting in Java refers to the **integration of scripting languages** (interpreted or dynamic languages like JavaScript, Groovy, Python, etc.) into Java applications using the **javax.script API**. It allows developers to:

- Execute scripts at runtime.
- Modify or extend application logic without recompiling Java code.
- Create dynamic and configurable applications.
- Support plugins or user-defined logic.

29.2 Java Scripting API (JSR 223)

The Java Scripting API was introduced in Java SE 6 as part of **javax.script** package.

Key Interfaces and Classes:

Class/Interface	Description
`ScriptEngineManager`	Creates and manages `ScriptEngine` instances.
`ScriptEngine`	Represents an interpreter for a specific scripting language.
`Bindings`	A map of key-value pairs passed to the scripting context.
`ScriptContext`	Contains script execution context such as input/output and variable scope.

29.3 JavaScript Engine in Java

29.3.1 Rhino and Nashorn

- **Rhino**: Originally the default JavaScript engine (developed by Mozilla) until Java 7.
- **Nashorn**: Introduced in Java 8 to replace Rhino. It provides improved performance and better integration with Java.

📝 **Note**: Nashorn has been deprecated in Java 11 and removed in Java 15, but it remains an important historical and educational tool.

29.3.2 Basic Nashorn Example

```
import javax.script.*;

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

        engine.eval("print('Hello from JavaScript!');");
    }
}
```

29.4 Working with Variables and Bindings

You can pass variables from Java to the script using the `Bindings` object:

```
Bindings bindings = engine.createBindings();
bindings.put("x", 10);
bindings.put("y", 20);
engine.setBindings(bindings, ScriptContext.ENGINE_SCOPE);
engine.eval("print('Sum = ' + (x + y));");
```

29.5 Calling Java from JavaScript

Nashorn supports calling Java methods/classes from scripts:

```
engine.eval("var File = Java.type('java.io.File');"
          + "var file = new File('test.txt');"
          + "print(file.getAbsolutePath());");
```

29.6 Invoking Script Functions from Java

You can invoke a script function defined in JavaScript using the `Invocable` interface:

```
String script =
    "function greet(name) { return 'Hello, ' + name; }";

engine.eval(script);

Invocable invocable = (Invocable) engine;
String result = (String) invocable.invokeFunction("greet", "Abraham");
System.out.println(result); // Output: Hello, Abraham
```

29.7 Scripting Use Cases in Java Applications

Use Case	Description
Dynamic Business Rules	Let users or admins write rules in a script file that Java can execute at runtime.
Scripting in IDEs or Tools	Tools like NetBeans or Eclipse use embedded scripting for plugin support.
Web Template Engines	Allow embedding scripting logic in HTML templates.
Testing and Prototyping	Quickly test features without full compilation cycles.
Plugins and Extensions	Applications can expose scripting hooks for customization.

29.8 Advantages of Scripting in Java

- **Flexibility**: Modify behavior without recompilation.
- **Rapid Prototyping**: Ideal for testing ideas quickly.
- **User Customization**: End-users can define custom logic.
- **Integration**: Easily combine compiled Java and dynamic scripting.

29.9 Challenges and Limitations

- **Performance**: Interpreted scripts are slower than compiled Java.
- **Security**: Executing arbitrary scripts can be risky; sandboxing is often required.
- **Maintenance**: Debugging scripts may be harder than Java code.
- **Deprecation of Nashorn**: Newer Java versions don't include Nashorn by default.

29.10 Alternatives to Nashorn

With Nashorn deprecated, here are modern alternatives:

Engine	Language	Notes
GraalVM	JavaScript, Python, Ruby, etc.	High-performance polyglot VM.
Jython	Python	Python interpreter for Java.
Groovy	Groovy	Seamlessly integrates with Java.
BeanShell	Java-like	Lightweight scripting.

29.11 Embedding Groovy as a Scripting Language

```
import org.codehaus.groovy.jsr223.GroovyScriptEngineImpl;

ScriptEngine engine = new GroovyScriptEngineImpl();
engine.eval("println 'Hello from Groovy!'");
```

29.12 Practical Example: Business Rule Engine

```
String script = "if (orderTotal > 1000) { discount = 10; } else { discount =
0; }";
Bindings bindings = engine.createBindings();
bindings.put("orderTotal", 1200);
engine.eval(script, bindings);
System.out.println("Discount: " + bindings.get("discount"));
```

29.13 Summary

Java's scripting support provides a powerful bridge between the compiled robustness of Java and the flexibility of interpreted scripting languages. While Nashorn served as the cornerstone for JavaScript integration, modern solutions like **GraalVM** are now preferred for multi-language scripting. The ability to execute scripts dynamically enables new architectures like plugin-based systems, configurable engines, and dynamic business rule processing.
