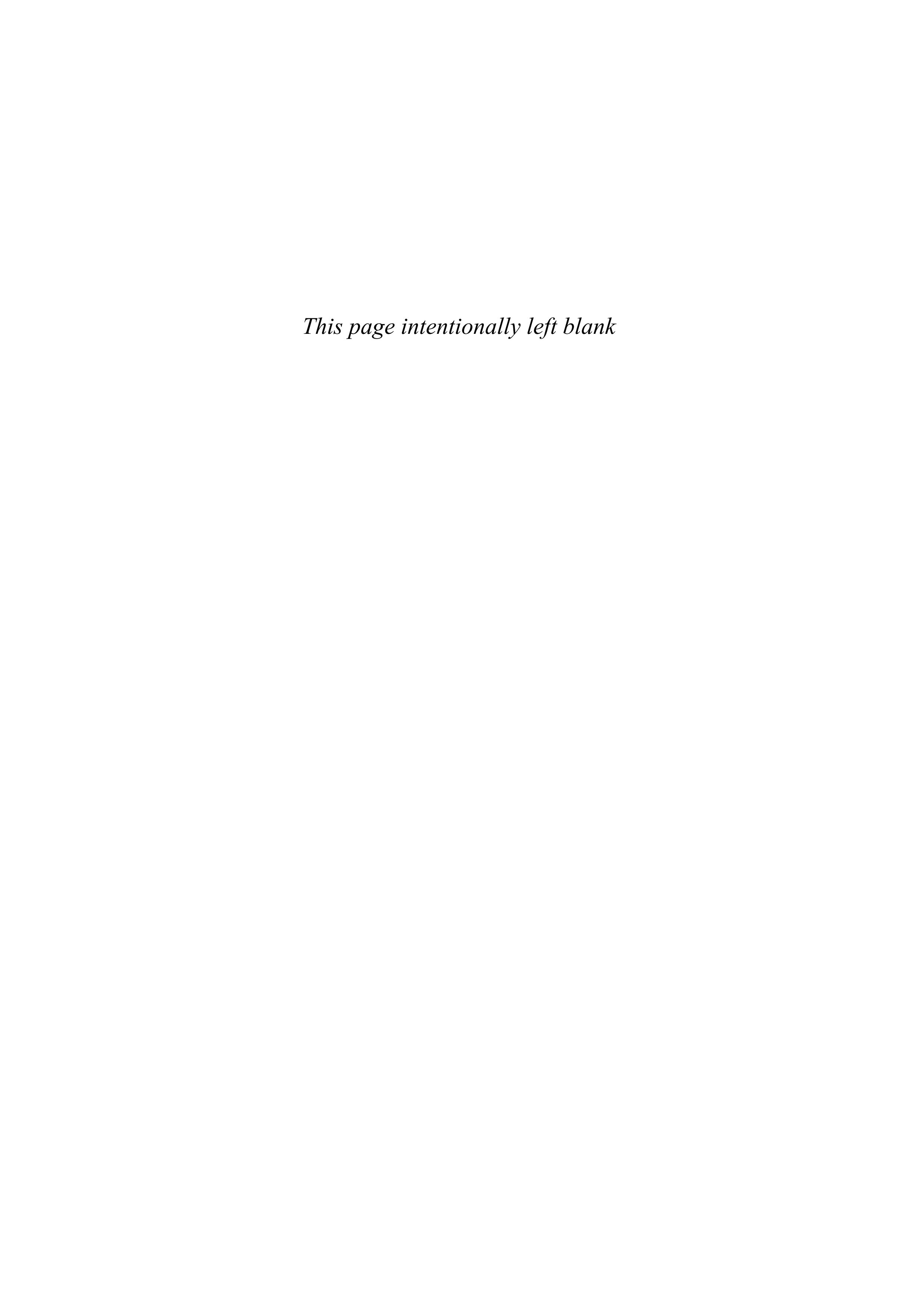
Scripting in Java™

Languages, Frameworks, and Patterns

DEJAN BOSANAC

Scripting in Java™

Languages, Frameworks, and Patterns



This page intentionally left blank

Scripting in Java™

Languages, Frameworks, and Patterns

Dejan Bosanac

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearsoned.com

Visit us on the Web: www.awprofessional.com

Library of Congress Cataloging-in-Publication Data

Bosanac, Dejan.
Scripting in java : languages, frameworks, and patterns / Dejan Bosanac.
p. cm.
ISBN 0-321-32193-6 (pbk. : alk. paper) 1. Java (Computer program language) 2. Programming languages (Electronic computers) I. Title.

QA76.73.J38B6715 2007
005.13'3—dc22

2007017654

Copyright © 2008 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

ISBN-13: 978-0-321-32193-0
ISBN-10: 0-321-32193-6
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, IN.
First printing August 2007

EDITOR-IN-CHIEF
Mark Taub

ACQUISITIONS EDITOR
Greg Doench

DEVELOPMENT EDITOR
Audrey Doyle

MANAGING EDITOR
Gina Kanouse

PROJECT EDITOR
Anne Goebel

COPY EDITOR
Geneil Breeze

INDEXER
Brad Herriman

PROOFREADER
Water Crest Publishing, Inc.

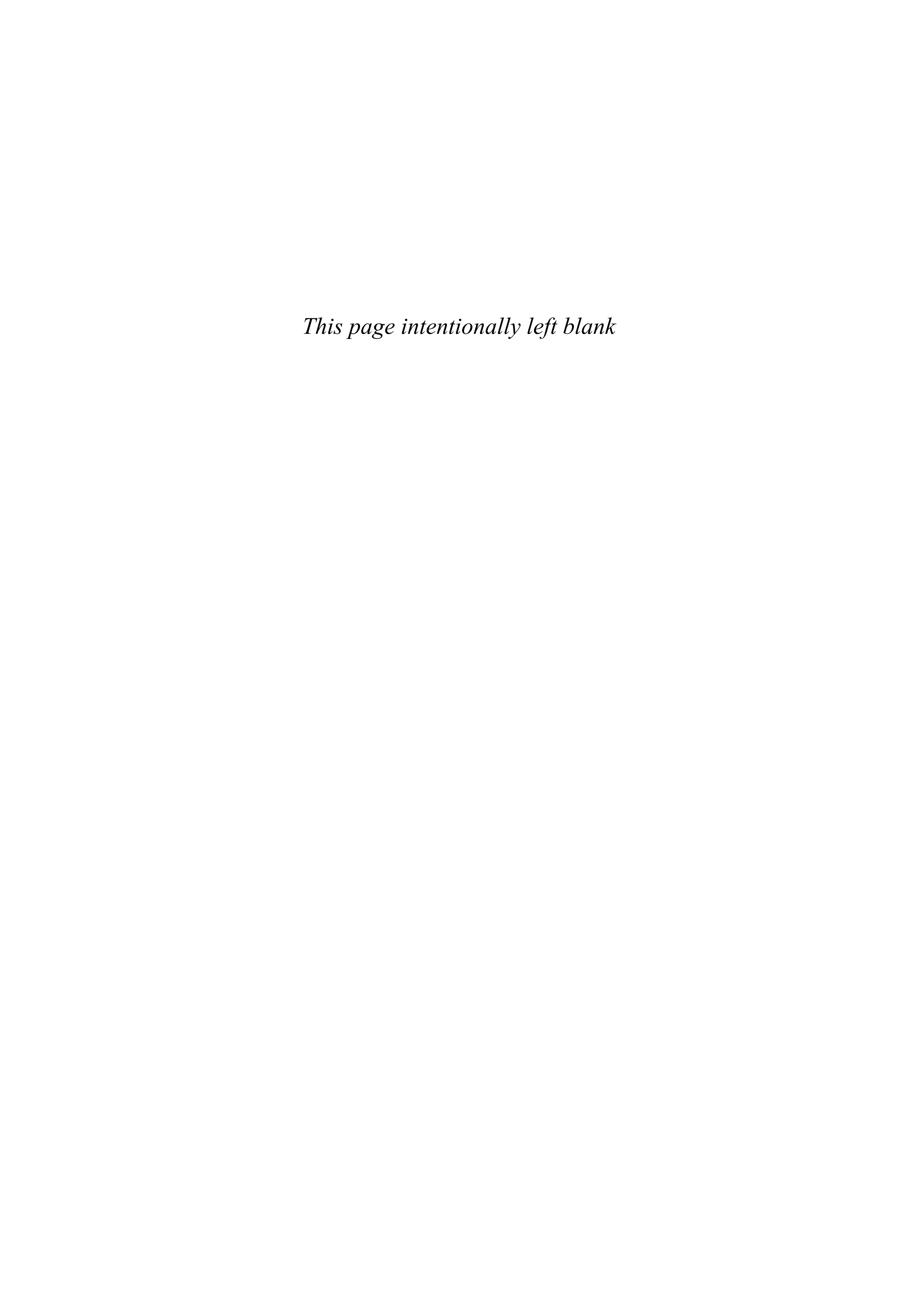
PUBLISHING COORDINATOR
Michelle Housley

COVER DESIGNER
Chuti Prasertsith

COMPOSITION
Bumpy Design

Dedication

To Ivana, for being so lovely


This page intentionally left blank

CONTENTS

PREFACE xvii

PART I 1

CHAPTER 1 INTRODUCTION TO SCRIPTING 3

BACKGROUND 4

DEFINITION OF A SCRIPTING LANGUAGE 8

COMPILERS VERSUS INTERPRETERS 8

SOURCE CODE IN PRODUCTION 12

TYPING STRATEGIES 13

DATA STRUCTURES 17

CODE AS DATA 19

SUMMARY 23

SCRIPTING LANGUAGES AND VIRTUAL MACHINES 24

A COMPARISON OF SCRIPTING AND SYSTEM PROGRAMMING 26

RUNTIME PERFORMANCE 26

DEVELOPMENT SPEED 28

ROBUSTNESS 29

MAINTENANCE 32

EXTREME PROGRAMMING 33

THE HYBRID APPROACH 35

A CASE FOR SCRIPTING 37

CONCLUSION 38

CHAPTER 2 APPROPRIATE APPLICATIONS FOR SCRIPTING LANGUAGES 39

WIRING 40

UNIX SHELL LANGUAGES 41

Perl 43
Tcl 43
Prototyping 44
Python 47
Customization 49
Visual Basic for Applications (VBA) 50
Software Development Support 51
Project Building 51
Testing 53
Administration and Management 55
User Interface Programming 58
Tk 58
Use Cases 59
Web Applications 59
Scripting and UNIX 68
Scripting in Games 68
Additional Characteristics 69
Embeddable 70
Extensible 70
Easy to Learn and Use 71
Conclusion 72

Part II 75

Chapter 3 Scripting Languages Inside the JVM 77
Under the Hood 80
Scripting Language Concepts 82
BeanShell 83
Getting Started 83
Basic Syntax 86
Loosely Typed Syntax 87

Syntax Flavors 88
Commands 91
Methods 91
Objects 92
Implementing Interfaces 93
Embedding with Java 94
Jython 98
Getting Started 98
Basic Syntax 101
Working with Java 103
Implementing Interfaces 105
Exception Handling 107
Embedding with Java 108
Conclusion 109
Rhino 110
Getting Started 110
Working with Java 111
Implementing Interfaces 112
JavaAdapter 114
Embedding with Java 114
Host Objects 117
Conclusion 120
Groovy 120
Other Scripting Languages 122
JRuby 122
Tcl/Java 122
JudoScript 122
ObjectScript 123
Conclusion 123

Chapter 4 Groovy 125

Why Groovy? 126

Installation 127

Running Groovy Scripts 127

Using the Interactive Shell 127

Using the Interactive Console 128

Evaluating the Script File 129

Compiling Groovy Scripts 130

Dependencies 131

Classpath 131

Ant Task 132

Script Structure 133

Command-Line Arguments 136

Language Syntax 137

Java Compatibility 137

Statements 138

Loose Typing 138

Type Juggling 140

Strings 143

GStrings 145

Regular Expressions 146

Collections 148

Logical Branching 154

Looping 156

Classes 159

Operator Overloading 162

GroovyBeans 165

Closures 168

System Operations 178

Files 178

Processes 182

Embedding with Java 184
Security 190
Conclusion 194

Chapter 5 Advanced Groovy Programming 195

GroovySQL 196
`groovy.sql.Sql` *198*
`groovy.sql.DataSet` *209*
Groovlets 212
Groovy Templates 220
GroovyMarkup 223
`groovy.xml.MarkupBuilder` *224*
`groovy.util.NodeBuilder` *227*
`groovy.xml.SaxBuilder` *230*
`groovy.xml.DomBuilder` *232*
`groovy.xml.Namespace` *234*
`groovy.util.BuilderSupport` *235*
Groovy and Swing 236
`TableLayout` *239*
`TableModel` *241*
Conclusion 243

Chapter 6 Bean Scripting Framework 245

Introduction to the Bean Scripting Framework 246
Getting Started 247
Basic Concepts 248
Architecture 248
Registration of Scripting Languages 249
Manager and Engine Initialization 252
Working with Scripts 253
Working with Script Files 257

METHODS AND FUNCTIONS 259

`call()` *259*

`apply()` *263*

DATA BINDING 264

REGISTERING BEANS *265*

DECLARING BEANS *268*

COMPILATION 270

APPLICATIONS 275

JSP *275*

XALAN-J (XSLT) *280*

CONCLUSION 288

PART III 289

CHAPTER 7 PRACTICAL SCRIPTING IN JAVA 291

UNIT TESTING 292

JUNIT BASICS *293*

THE `GroovyTestCase` *CLASS* *296*

ASSERTION METHODS *297*

TEST SUITES *300*

SCRIPTS AS UNIT TEST CASES *303*

SUMMARY *304*

INTERACTIVE DEBUGGING 304

BUILD TOOLS (ANT SCRIPTING) 309

BSF SUPPORT *313*

GROOVYMARKUP (ANTBUILDER) *316*

SUMMARY *322*

SHELL SCRIPTING 323

CLASSPATH *324*

EXAMPLE *325*

ADMINISTRATION AND MANAGEMENT 328

CONCLUSION 334

Chapter 8 Scripting Patterns 335

Scripted Components Pattern 337

Problem 337

Solution 338

Consequences 339

Sample Code 340

Related Patterns 341

Mediator Pattern (Glue Code Pattern) 341

Problem 341

Solution 342

Consequences 345

Sample Code 345

Related Patterns 354

Script Object Factory Pattern 354

Problem 355

Solution 355

Consequences 356

Sample Code 356

Related Patterns 359

Observer (Broadcasters) Pattern 359

Problem 359

Solution 360

Consequences 362

Sample Code 362

Related Patterns 369

Extension Point Pattern 369

Problem 369

Solution 370

Consequences 370

Sample Code 371

Related Patterns 374

ACTIVE FILE PATTERN 375
PROBLEM *375*
SOLUTION *375*
CONSEQUENCES *375*
SAMPLE CODE *376*
CONCLUSION 380

PART IV 383

CHAPTER 9 SCRIPTING API 385
MOTIVATION AND HISTORY 386
INTRODUCTION 388
GETTING STARTED 390
ARCHITECTURE 391
DISCOVERY MECHANISM 391
ENGINE METADATA 393
CREATING AND REGISTERING SCRIPTING ENGINES 395
CREATION METHODS *396*
REGISTRATION METHODS *399*
EVALUATION 400
`ScriptException` 403
BINDING 404
ENGINE SCOPE *405*
GLOBAL SCOPE *411*
SCRIPT CONTEXT *416*
CODE GENERATION 428
OUTPUT STATEMENT *429*
METHOD CALL SYNTAX *429*
PROGRAM *431*
ADDITIONAL ENGINE INTERFACES 432
INVOCABLE *432*
COMPILABLE *437*

Threading 440

Dynamic Bindings 442

Conclusion 445

Chapter 10 Web Scripting Framework 447

Architecture 448

Context 448

Servlet 449

Interaction 451

Getting Started 453

Configuration 456

Disable Scripting 456

Script Directory 457

Script Methods 458

Allow Languages 459

Display Result 460

Bindings 462

Application 462

Request 464

Response 468

Servlet 468

Include Method 469

Forward Method 471

Session Sharing 473

Language Tags 478

Threading Issues 481

Architectural Challenges 482

Integration of Java and PHP Applications 482

Java Business Logic in PHP Web Applications 484

PHP Views in Java Web Applications 487

Conclusion 488

PART V 489

APPENDIX A GROOVY INSTALLATION 491

DOWNLOAD INSTRUCTIONS 491

INSTALLING GROOVY 492

CONFIGURING GROOVY 492

TESTING GROOVY 492

APPENDIX B GROOVY IDE SUPPORT 495

INSTALLATION 495

USAGE 497

APPENDIX C INSTALLING JSR 223 499

REQUIREMENTS 500

INSTALLATION 500

INDEX 503

PREFACE

Java is an excellent object-oriented programming language. It has provided many benefits to software developers, including a good object-oriented approach, implicit memory management, and dynamic linking, among others. These language characteristics are one of the main reasons for Java's popularity and wide acceptance.

But Java is much more than a programming language; it's a whole development platform. This means that it comes with a runtime environment (JRE), which provides the virtual machine, and the standardized application programming interfaces (APIs) that help developers accomplish most of their desired tasks. The main advantages of this integrated runtime environment are its true platform independence and simplification of software development.

On the other hand, scripting languages have played an important role in the information technology infrastructure for many years. They have been used for all kinds of tasks, ranging from job automation to prototyping and implementation of complex software projects.

Therefore, we can conclude that the Java development platform can also benefit from scripting concepts and languages. Java developers can use scripting languages in areas proven to be most suitable for this technology. This synergy of the Java platform and scripting languages, as we will see, adds an extra quality to the overall software development process.

In this book, I describe the concepts behind scripting languages, summarize solutions available to Java developers, and explore use cases and design patterns for applying scripting languages in Java applications.

How This Book Is Organized

This book consists of five logical parts.

Part I

The first part of the book comprises two chapters that describe scripting languages in general:

- **Chapter 1, "Introduction to Scripting"**—Here I define the basic characteristics of scripting languages and compare them to system programming languages.

- **Chapter 2, "Appropriate Applications for Scripting Languages"**—In this chapter, I explain the role of traditional (native) scripting languages in the overall information technology infrastructure. I also discuss tasks for which scripting languages have been used in various systems over time.

Part II

After discussing the basic concepts and uses of scripting languages, we are ready to focus on real technologies and solutions for the Java platform. This part of the book contains the following chapters:

- **Chapter 3, "Scripting Languages Inside the JVM"**—I begin this chapter by covering the basic elements of the Java platform and explaining where scripting languages fit into it. After that, I describe the main features of three popular scripting languages available for the Java Virtual Machine (JVM)—BeanShell, JavaScript, and Python—and how they can be used to interact with Java applications. At the end of this chapter, I describe other solutions available for Java developers.
- **Chapter 4, "Groovy"**—Here I discuss the Groovy scripting language in detail. I cover its Java-like syntax and all the scripting concepts built into this language, and I discuss Groovy's integration with Java, as well as some security-related issues.
- **Chapter 5, "Advanced Groovy Programming"**—In this chapter, I cover some of the Groovy extensions that can aid in day-to-day programming tasks. I also explain how Java programmers can access databases, create and process XML documents, and easily create simple Web applications and swing user interfaces, using the scripting-specific features in Groovy covered in Chapter 4.
- **Chapter 6, "Bean Scripting Framework"**—In this chapter, I describe the general Java scripting framework. In addition to explaining how to implement general support in your project for any compliant scripting language, I also discuss some basic abstractions implemented in the Bean Scripting Framework (BSF) and show some examples of successful uses.

Part III

This part of the book focuses primarily on the use of scripting languages in real Java projects:

- **Chapter 7, "Practical Scripting in Java"**—Here I cover topics related to the use of scripting for everyday programming tasks, such as unit testing, interactive debugging, and project building, among others.
- **Chapter 8, "Scripting Patterns"**—In this chapter, I discuss Java application design patterns that involve scripting languages. I show how you can use scripts to implement some parts of traditional design patterns and introduce some new design patterns specific only to the scripting environment. I also discuss the pros and cons of these design patterns, as well as their purpose.

Part IV

In the final part of this book, I cover the "Scripting for the Java Platform" specification, which was created according to the Java Specification Request (JSR) 223. Specifically, I cover two APIs defined by the specification:

- **Chapter 9, "Scripting API"**—Here I cover the Scripting API, the standardized general scripting framework for the Java platform. The purpose of this framework is the same as that of the Bean Scripting Framework, but the Scripting API brings many new features that the modern scripting framework needs. The Scripting API is a standard part of the Java platform with the release of Mustang (Java SE 6).
- **Chapter 10, "Web Scripting Framework"**—In this chapter, I discuss the Web Scripting Framework, a framework built on top of the Scripting API and created to enable scripting languages to generate Web content inside a servlet container. I explain how native scripting languages, such as PHP, can be synergized with the Java platform to bring more flexibility in Web application development.

Part V

At the end of the book, you can find a section comprising three appendixes. The main purpose of these appendixes is to provide the technical details about installation and use of certain technologies described in the book:

- **Appendix A, "Groovy Installation"**—In this appendix, I describe how to install, build, and configure the Groovy scripting language. A working installation of the Groovy interpreter is needed to run the code samples from the text.

- **Appendix B, "Groovy IDE Support"**—In this appendix, I provide instructions on how to install general Groovy support for Integrated Development Environments (IDEs).
- **Appendix C, "Installing JSR 223"**—Here I describe how to install the reference implementation (RI) of the JSR 223, which is needed to run examples from Chapter 10.

I hope you'll enjoy reading the book.

About the Web Site

This book is extended with a Web site at www.scriptinginjava.net where you can find the following:

- Source codes of all examples shown in the book available for download
- Book news, updates, and additions
- News and information related to this field of software development

Acknowledgments

I would like to thank my family, friends, and colleagues for endless patience and support during the writing of this book. I'm also grateful to the people from Addison-Wesley for believing in this material and making an excellent atmosphere to work in, especially my editors, Greg Doench and Ann Sellers. I would also like to thank all technical reviewers, especially George Jempty, Kevin Davis, and Rich Rosen. They have provided valuable feedback and helped me keep my focus when I was stranded. Without Audrey Doyle, this material would be much harder to read. Thank you for helping me shape the manuscript.

Finally, this book wouldn't be possible without all developers contributing their time to projects covered by this material.

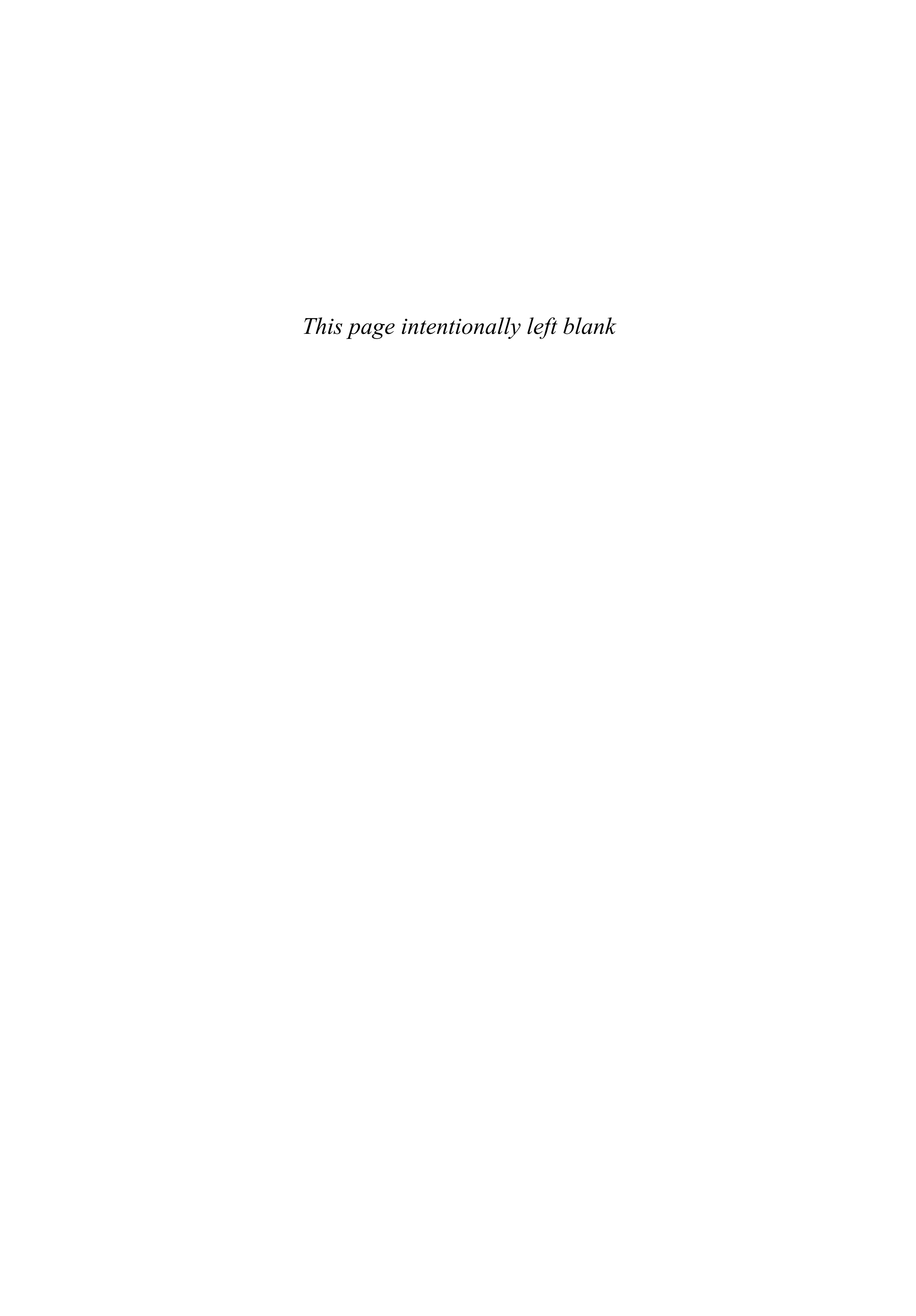
About the Author

Dejan Bosanac is a professional software developer and technology consultant. He is focused on the integration and interoperability of different technologies, especially ones related to Java and the Web. Dejan spent a number of years in development of complex software projects, ranging from highly trafficked Web sites to enterprise applications, and was a member of the JSR 223 Expert Group.

PART I

CHAPTER 1 Introduction to Scripting

CHAPTER 2 Appropriate Applications for Scripting Languages



This page intentionally left blank

Chapter 1

Introduction to Scripting

The main topic of this book is the synergy of scripting technologies and the Java platform. I describe projects Java developers can use to create a more powerful development environment, and some of the practices that make scripting useful.

Before I start to discuss the application of scripting in the Java world, I summarize some of the theory behind scripting in general and its use in information technology infrastructure. This is the topic of the first two chapters of the book, and it gives us a better perspective of scripting technology as well as how this technology can be useful within the Java platform.

To begin, we must define what scripting languages are and describe their characteristics. Their characteristics greatly determine the roles in which they could (should) be used. In this chapter, I explain what the term *scripting language* means and discuss their basic characteristics.

At the end of this chapter, I discuss the differences between scripting and system-programming languages and how these differences make them suitable for certain roles in development.

Background

The definition of a scripting language is fuzzy and sometimes inconsistent with how scripting languages are used in the real world, so it is a good idea to summarize some of the basic concepts about programming and computing in general. This summary provides a foundation necessary to define scripting languages and discuss their characteristics.

Let's start from the beginning. Processors execute *machine instructions*, which operate on data either in the processors' registers or in external memory. Put simply, a machine instruction is made up of a sequence of binary digits (0s and 1s) and is specific to the particular processor on which it runs. Machine instructions consist of the *operation code* telling the processor what operation it should perform, and *operands* representing the data on which the operation should be performed.

For example, consider the simple operation of adding a value contained in one register to the value contained in another. Now let's imagine a simple processor with an 8-bit instruction set, where the first 5 bits represent the operation code (say, 00111 for register value addition), and the registers are addressed by a 3-bit pattern. We can write this simple example as follows:

```
00111 001 010
```

In this example, I used `001` and `010` to address registers number one and two (`R1` and `R2`, respectively) of the processor.

This basic method of computing has been well known for decades, and I'm sure you are familiar with it. Various kinds of processors have different strategies regarding how their instruction sets should look (RISC or CISC architecture), but from the software developer's point of view, the only important fact is the processor is capable of executing only binary instructions. No matter what programming language is used, the resulting application is a sequence of machine instructions executed by the processor.

What has been changing over time is how people create the order in which the machine instructions are executed. This ordered sequence of machine instructions is called a *computer program*. As hardware is becoming more affordable and more powerful, users' expectations rise. The whole purpose of software development as a science discipline is to provide mechanisms enabling developers to craft more complex applications with the same (or even less) effort as before.

A specific processor's instruction set is called its *machine language*. Machine languages are classified as first-generation programming languages. Programs written in this way are usually very fast because they are optimized for the particular processor's architecture. But despite this benefit, it is hard (if not impossible) for humans to write large and secure applications in machine languages because humans are not good at dealing with large sequences of 0s and 1s.

In an attempt to solve this problem, developers began creating symbols for certain binary patterns, and with this, *assembly languages* were introduced. Assembly languages are *second-generation programming languages*. The instructions in assembly languages are just one level above machine instructions, in that they replace binary digits with easy-to-remember keywords such as `ADD`, `SUB`, and so on. As such, you can rewrite the preceding simple instruction example in assembly language as follows:

```
ADD R1, R2
```

In this example, the `ADD` keyword represents the operation code of the instruction, and `R1` and `R2` define the registers involved in the operation. Even if you observe just this simple example, it is obvious assembly languages made programs easier for humans to read and thus enabled creation of more complex applications.

Although they are much more human-oriented, however, second-generation languages do not extend processor capabilities by any means.

Enter *high-level languages*, which allow developers to express themselves in higher-level, semantic forms. As you might have guessed, these languages are referred to as *third-generation programming languages*. High-level languages provide various powerful loops, data structures, objects, and so on, making it much easier to craft many applications with them.

Over time, a diverse array of high-level programming languages were introduced, and their characteristics varied a great deal. Some of these characteristics categorize programming languages as scripting (or dynamic) languages, as we see in the coming sections.

Also, there is a difference in how programming languages are executed on the host machine. Usually, *compilers* translate high-level language constructs into machine instructions that reside in memory. Although programs written in this way initially were slightly less efficient than programs written in assembly language because of early compilers' inability to use system resources efficiently, as time passed compilers and machines improved, making system-programming languages superior to assembly languages. Eventually, high-level languages became popular in a wide range of development areas, from business applications and games to communications software and operating system implementations.

But there is another way to transform high-level semantic constructs into machine instructions, and that is to interpret them as they are executed. This way, your applications reside in scripts, in their original form, and the constructs are transformed at runtime by a program called an *interpreter*. Basically, you are executing the interpreter that reads statements of your application and then executes them. Called *scripting* or *dynamic languages*, such languages offer an even higher level of abstraction than that offered by system-programming languages, and we discuss them in detail later in this chapter.

Languages with these characteristics are a natural fit for certain tasks, such as process automation, system administration, and gluing existing software components together; in short, anywhere the strict syntax and constraints introduced by system-programming languages were getting in the way

between developers and their jobs. A description of the usual roles of scripting languages is a focus of Chapter 2, "Appropriate Applications for Scripting Languages."

But what does all this have to do with you as a Java developer? To answer this question, let's first briefly summarize the history of the Java platform. As platforms became more diverse, it became increasingly difficult for developers to write software that can run on the majority of available systems. This is when Sun Microsystems developed Java, which offers "write once, run anywhere" simplicity.

The main idea behind the Java platform was to implement a virtual processor as a software component, called a *virtual machine*. When we have such a virtual machine, we can write and compile the code for that processor, instead of the specific hardware platform or operating system. The output of this compilation process is called *bytecode*, and it practically represents the machine code of the targeted virtual machine. When the application is executed, the virtual machine is started, and the bytecode is interpreted. It is obvious an application developed in this way can run on any platform with an appropriate virtual machine installed. This approach to software development found many interesting uses.

The main motivation for the invention of the Java platform was to create an environment for the development of easy, portable, network-aware client software. But mostly due to performance penalties introduced by the virtual machine, Java is now best suited in the area of server software development. It is clear as personal computers increase in speed, more desktop applications are being written in Java. This trend only continues.

One of the basic requirements of a scripting language is to have an interpreter or some kind of virtual machine. The Java platform comes with the Java Virtual Machine (JVM), which enables it to be a host to various scripting languages. There is a growing interest in this area today in the Java community. Few projects exist that are trying to provide Java developers with the same power developers of traditional scripting languages have. Also, there is a way to execute your existing application written in a dynamic language such as Python inside the JVM and integrate it with another Java application or module.

This is what we discuss in this book. We take a scripting approach to programming, while discussing all the strengths and weaknesses of this approach, how to best use scripts in an application architecture, and what tools are available today inside the JVM.

Definition of a Scripting Language

There are many definitions of the term *scripting language*, and every definition you can find does not fully match some of the languages known to be representatives of scripting languages. Some people categorize languages by their purpose and others by their features and the concepts they introduce. In this chapter, we discuss all the characteristics defining a scripting language. In Chapter 2, we categorize scripting languages based on their role in the development process.

Compilers Versus Interpreters

Strictly speaking, an *interpreter* is a computer program that executes other high-level programs line by line. Languages executed only by interpreters are called *interpreted languages*.

To better understand the differences between compilers and interpreters, let's take a brief look at compiler architecture (see Figure 1.1).

As you can see in Figure 1.1, translating source code to machine code involves several steps:

1. First, the source code (which is in textual form) is read character by character. The scanner groups individual characters into valid language constructs (such as variables, reserved words, and so on), called *tokens*.
2. The tokens are passed to the parser, which checks that the correct language syntax is being used in the program. In this step, the program is converted to its parse tree representation.
3. Semantic analysis performs type checking. Type checking validates that all variables, functions, and so on, in

the source program have been used consistently with their definitions. The result of this phase is intermediate representation (IR) code.

4. Next, the optimizer (optionally) tries to make equivalent but improved IR code.
5. In the final step, the code generator creates target machine code from the optimized IR code. The generated machine code is written as an object file.

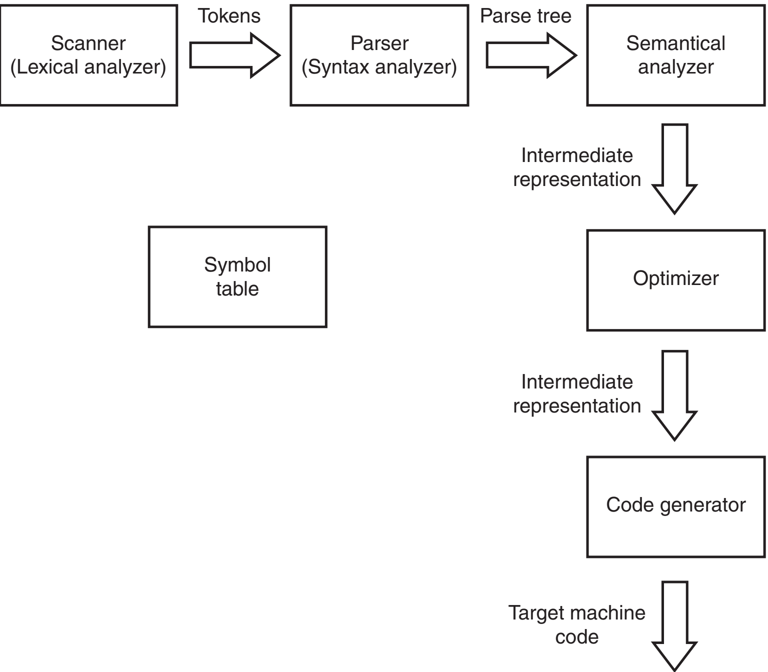


FIGURE 1.1 Compiler architecture

To create one executable file, a linking phase is necessary. The linker takes several object files and libraries, resolves all external references, and creates one executable object file. When such a compiled program is executed, it has complete control of its execution.

Unlike compilers, interpreters handle programs as data that can be manipulated in any suitable way (see Figure 1.2).

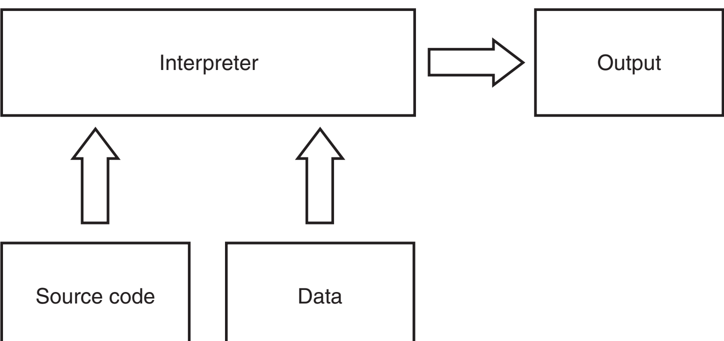


FIGURE 1.2 Interpreter architecture

As you can see in Figure 1.2, the interpreter, not the user program, controls program execution. Thus, we can say the user program is passive in this case. So, to run an interpreted program on a host, both the source code and a suitable interpreter must be available. The presence of the program source (script) is the reason why some developers associate interpreted languages with scripting languages. In the same manner, compiled languages are usually associated with system-programming languages.

Interpreters usually support two modes of operation. In the first mode, the script file (with the source code) is passed to the interpreter. This is the most common way of distributing scripted programs. In the second, the interpreter is run in interactive mode. This mode enables the developer to enter program statements line by line, seeing the result of the execution after every statement. Source code is not saved to the file. This mode is important for initial system debugging, as we see later in the book.

In the following sections, I provide more details on the strengths and weaknesses of using compilers and interpreters. For now, here are some clear drawbacks of both approaches important for our further discussion:

- It is obvious compiled programs usually run faster than interpreted ones. This is because with compiled programs, no high-level code analysis is being done during runtime.

- An interpreter enables the modification of a user program as it runs, which enables interactive debugging capability. In general, interpreted programs are much easier to debug because most interpreters point directly to errors in the source code.
- Interpreters introduce a certain level of machine independence because no specific machine code is generated.
- The important thing from a scripting point of view, as we see in a moment, is interpreters allow the variable type to change dynamically. Because the user program is reexamined constantly during execution, variables do not need to have fixed types. This is much harder to accomplish with compilers because semantic analysis is done at compile time.

From this list, we can conclude interpreters are better suited for the development process, and compiled programs are better suited for production use. Because of this, for some languages, you can find both an interpreter and a compiler. This means you can reap all the benefits of interpreters in the development phase and then compile a final version of the program for a specific platform to gain better performance.

Many of today's interpreted languages are not interpreted purely. Rather, they use a hybrid compiler-interpreter approach, as shown in Figure 1.3.

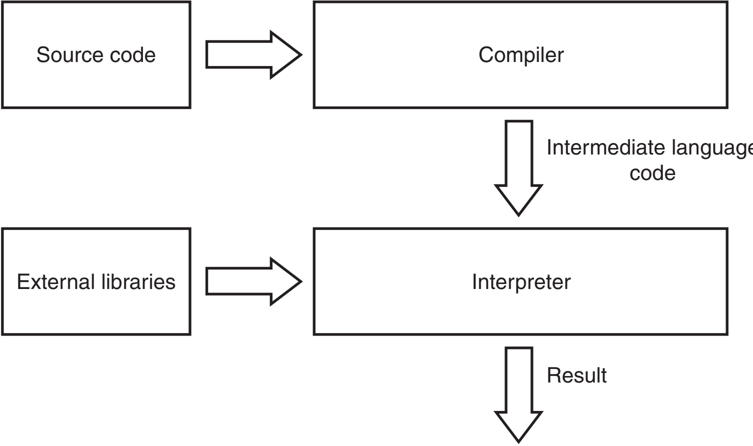


FIGURE 1.3 Hybrid compiler-interpreter architecture

In this model, the source code is first compiled to some intermediate code (such as Java bytecode), which is then interpreted. This intermediate code is usually designed to be very compact (it has been compressed and optimized). Also, this language is not tied to any specific machine. It is designed for some kind of virtual machine, which could be implemented in software. Basically, the virtual machine represents some kind of processor, whereas this intermediate code (bytecode) could be seen as a machine language for this processor.

This hybrid approach is a compromise between pure interpreted and compiled languages, due to the following characteristics:

- Because the bytecode is optimized and compact, interpreting overhead is minimized compared with purely interpreted languages.
- The platform independence of interpreted languages is inherited from purely interpreted languages because the intermediate code could be executed on any host with a suitable virtual machine.

Lately, just-in-time compiler technology has been introduced, which allows developers to compile bytecode to machine-specific code to gain performance similar to compiled languages. I mention this technology throughout the book, where applicable.

Source Code in Production

As some people have pointed out, you should use a scripting language to write user-readable and modifiable programs that perform simple operations and control the execution of other programs. In this scenario, source code should be available in the production system at runtime, so programs are delivered not in object code, but in plain text files (scripts) in their original source. From our previous discussion of interpreters, it is obvious this holds true for purely interpreted languages. Because scripting languages are interpreted, we can say this rule applies to them as well. But because some of them use a hybrid compilation-interpretation strategy, it is possible to deliver the

program in intermediate bytecode form. The presence of the bytecode improves execution speed because no compilation process is required. The usual approach is to deliver necessary libraries in the bytecode and not the program itself. This way, execution speed is improved, and the program source is still readable in production. Some of the compiler-interpreter languages cache in the file the bytecode for the script on its first execution. On every following script execution, if the source hasn't been changed, the interpreter uses the cached bytecode, improving the startup speed required to execute the script.

As such, the presence of source code in the production environment is one of the characteristics of scripting languages, although you can omit it for performance reasons or if you want to keep your source code secret.

Typing Strategies

Before I start a discussion on typing strategies implemented in different programming languages, I have to explain what types are.

There is no simple way to explain what typing is because its definition depends on the context in which it is used. Also, a whole branch of mathematics is dedicated to this issue. It is called *type theory*, and its proponents have the following saying, which emphasizes their attitude toward the importance of this topic:

Design the type system correctly, and the language will design itself.

To put it simply, types are metadata that describe the data stored in some variable. Types specify what values can be stored in certain variables, as well as the operations that can be performed on them.

Type constraints determine how we can handle and operate a certain variable. For example, what happens when you add the values of one variable to those of another depends on whether the variables are integers, floats, Booleans, or strings. A programming language's type system could classify the value

`hello` as a string and the value `7` as a number. Whether you can mix strings with numbers in this language depends on the language's type policy.

Some types are *native* (or *primitive*), meaning they are built into the language. The usual representatives of this type category are Booleans, integers, floats, characters, and even strings in some languages. These types have no visible internal structure.

Other types are *composite,* and are constructed of primitive types. In this category, we have structures and various so-called container types, such as lists, maps, and sets. In some languages, *string* is defined as a list of characters, so it can be categorized as a composite type.

In object-oriented languages, developers got the opportunity to create their own types, also known as *classes*. This type category is called *user-defined types*. The big difference between structures and classes is with classes, you define not just the structure of your complex data, but also the behavior and possible operations you can perform with it. This categorizes every class as a single type, where structures (in C, for example) are one type.

Type systems provide the following major benefits:

- **Safety**—Type systems are designed to catch the majority of type-misuse mistakes made by developers. In other words, types make it practically impossible to code some operations that cannot be valid in a certain context.
- **Optimization**—As I already mentioned, languages that employ static typing result in programs with better-optimized machine code. That is because early type checks provide useful information to the compiler, making it easier to allocate optimized space in memory for a certain variable. For example, there is a great difference in memory usage when you are dealing with a Boolean variable versus a variable containing some random text.
- **Abstraction**—Types allow developers to make better abstractions in their code, enabling them to think about programs at a higher level of abstraction, not bothering

with low-level implementation of those types. The most obvious example of this is in the way developers deal with strings. It is much more useful to think of a string as a text value rather than as a byte array.

- **Modularity**—Types allow developers to create application programming interfaces (APIs) for the subsystems used to build applications. Typing localizes the definitions required for interoperability of subsystems and prevents inconsistencies when those subsystems communicate.
- **Documentation**—Use of types in languages can improve the overall documentation of the code. For example, a declaration that some method's arguments are of a specific type documents how that method can be used. The same is true for return values of methods and variables.

Now that we know the basic concepts of types and typing systems, we can discuss the type strategies implemented in various languages. We also discuss how the choice of implemented typing system defines languages as either scripting (dynamic) or static.

DYNAMIC TYPING

The type-checking process verifies that the constraints introduced by types are being respected. System-programming languages traditionally used to do type checking at compile time. This is referred to as *static typing.*

Scripting languages force another approach to typing. With this approach, type checking is done at runtime. One obvious consequence of runtime checking is all errors caused by inappropriate use of a type are triggered at runtime. Consider the following example:

```
x = 7
y = “hello world”
z = x + y
```

This code snippet defines an integer variable, `x`, and a string variable, `y`, and then tries to assign a value for the `z` variable that is the sum of the `x` and `y` values. If the language has not

defined an operator, +, for these two types, different things happen depending on whether the language is statically or dynamically typed. If the language was statically typed, this problem would be discovered at compile time, so the developer would be notified of it and forced to fix it before even being able to run the program. If the language was dynamically typed, the program would be executable, but when it tried to execute this problematic line, a runtime error would be triggered.

Dynamic typing usually allows a variable to change type during program execution. For example, the following code would generate a compile-time error in most statically typed programming languages:

```
x = 7
x = "Hello world"
```

On the other hand, this code would be legal in a purely dynamic typing language. This is simply because the type is not being misused here.

Dynamic typing is usually implemented by tagging the variables. For example, in our previous code snippet, the value of variable `x` after the first line would be internally represented as a pair (`7`, `number`). After the second line, the value would be internally represented as a pair ("`Hello world`", `string`). When the operation is executed on the variable, the type is checked and a runtime error is triggered if the misuse is discovered. Because no misuse is detected in the previous example, the code snippet runs without raising any errors.

I comprehensively discuss the pros and cons of these approaches later in this chapter, but for now, it is important to note a key benefit of dynamic typing from the developer's point of view. Programs written in dynamically typed languages tend to be much shorter than equivalent solutions written in statically typed languages. This is an implication of the fact that developers have much more freedom in terms of expressing their ideas when they are not constrained by a strict type system.

WEAK TYPING

There is yet another categorization of programming-language typing strategy. Some languages raise an error when a programmer tries to execute an operation on variables whose types are not suitable for that operation (type misuse). These languages are called *strongly typed languages*. On the other hand, *weakly typed languages* implicitly cast (convert) a variable to a suitable type before the operation takes place.

To clarify this, let's take a look at our first example of summing a number and string variable. In a strongly typed environment, which most system-programming languages deploy, this operation results in a compile-time error if no operator is defined for these types. In a weakly typed language, the integer value usually would be converted to its string representative (`7` in this case) and concatenated to the other string value (supposing that the + operator represents string concatenation in this case). The result would be a `z` variable with the "`7HelloWorld`" value and the string type.

Most scripting languages tend to be dynamic and weakly typed, but not all of them use these policies. For example, Python, a popular scripting language, employs dynamic typing, but it is strongly typed. We discuss in more detail the strengths and weaknesses of these typing approaches, and how they can fit into the overall system architecture, later in this chapter and in Chapter 2.

Data Structures

For successful completion of common programming tasks, developers usually need to use different complex data structures. The presence of language mechanisms for easy handling of complex data structures is in direct connection to developers' efficiency.

Scripting languages generally provide more powerful and flexible built-in data types than traditional system-programming languages. It is natural to see data structures such as lists, sets, maps, and so on, as native data types in such languages.

Of course, it is possible to implement an arbitrary data structure in any language, but the point is these data structures are embedded natively in language syntax making them much easier to learn and use. Also, without this standard implementation, novice developers are often tempted to create their own solution that is usually not robust enough for production use.

As an example, let's look at Python, a popular dynamic language with lists and maps (also called dictionaries) as its native language type. You can use these structures with other language constructs, such as a `for` loop, for instance. Look at the following example of defining and iterating a simple list:

```
list = ["Mike", "Joe", "Bruce"]
for item in list :
    print item
```

As you can see, the Python code used in this example to define a list is short and natural. But more important is the `for` loop, which is designed to naturally traverse this kind of data. Both of these features make for a comfortable programming environment and thus save some time for developers.

Java developers may argue that Java collections provide the same capability, but prior to J2SE 1.5, the equivalent Java code would look like this:

```
String[] arr = new String[]{"Mike", "Joe", "Bruce"};
List list = Arrays.asList(arr);
for (Iterator it = list.iterator(); it.hasNext(); ) {
      System.out.println(it.next());
}
```

Even for this simple example, the Java code is almost twice as long as and is much harder to read than the equivalent Python code. In J2SE 1.5, Java got some features that brought it closer to these scripting concepts. With the more flexible `for` loop, you could rewrite the preceding example as follows:

```
String[] arr = new String[]{"Mike", "Joe", "Bruce"};
List list = Arrays.asList(arr);
for (String item : list) {
      System.out.println(item);
}
```

With this in mind, we can conclude data structures are an important part of programming, and therefore native language support for commonly used structures could improve developers' productivity. Many scripting languages come with flexible, built-in data structures, which is one of the reasons why they are often categorized as "human-oriented."

Code as Data

The code and data in compiled system programming languages are two distinct concepts. Scripting languages, however, attempt to make them more similar. As I said earlier, programs (code) in scripting languages are kept in plain text form. Language interpreters naturally treat them as ordinary strings.

EVALUATION

It is not unusual for the commands (built-in functions) in scripting languages to evaluate a string (data) as language expression (code). For example, in Python, you can use the `eval()` function for this purpose:

```
x = 9
eval("print x + 7")
```

This code prints `16` on execution, meaning the value of the variable `x` is embedded into the string, which is evaluated as a regular Python program.

More important is the fact that scripted programs can generate new programs and execute them "on the fly". Look at the following Python example:

```
temp = open("temp.py", "w")
temp.write("print x + 7")
temp.close()
x = 9
execfile("temp.py")
```

In this example, we created a file called `temp.py`, and we wrote a Python expression in it. At the end of the snippet, the `execfile()` command executed the file, at which point `16` was displayed on the console.

This concept is natural to interpreted languages because the interpreter is already running on the given host executing the current script. Evaluation of the script generated at runtime is not different from evaluation of other regular programs. On the other hand, for compiled languages this could be a challenging task. That is because a compile/link phase is introduced during conversion of the source code to the executable program. With interpreted languages, the interpreter must be present in the production environment, and with compiled languages, the compiler (and linker) is usually not part of the production environment.

CLOSURES

Scripting languages also introduce a mechanism for passing blocks of code as method arguments. This mechanism is called a *closure*. A good way to demonstrate closures is to use methods to select items in a list that meet certain criteria.

Imagine a list of integer values. We want to select only those values greater than some threshold value. In Ruby, a scripting language that supports closures, we can write something like this:

```
threshold = 10
newList = orig.select {|item| item > threshold}
```

The `select()` method of the collection object accepts a closure, defined between the `{}`, as an argument. If parameters must be passed, they can be defined between the `||`. In this example, the `select()` method iterates over the collection, passing each item to the closure (as an `item` parameter) and returning a collection of items for which the closure returned `true`.

Another thing worth noting in this example is closures can refer to variables visible in the scope in which the closure is created. That's why we could use the global `threshold` value in the closure.

Closures in scripting languages are not different from any other data type, meaning methods can accept them as parameters and return them as results.

FUNCTIONS AS METHOD ARGUMENTS

Many scripting languages, even object-oriented ones, introduce standalone functions as so-called "first-class language citizens." Even if you do not have true support for closures, you can pass your functions as method arguments.

The Python language, for example, defines a `filter()` function that accepts a list and the function to be executed on every item in the list:

```
def over(item) :
    threshold = 10
    return item > threshold

newList = filter(over, orig)
```

In this example, we defined the `over()` function, which basically does the same job as our closure from the previous example. Next, we called the `filter()` function and passed the `over()` function as the second argument. Even though this mechanism is not as convenient as closures are, it serves its purpose well (and that is to pass blocks of code as data around the application).

Of course, you can achieve similar functionality in other nonscripting languages. For example, Java developers have the concept of anonymous inner classes serving the same purpose. Let's implement a similar solution using this approach:

```
package net.scriptinginjava.ch1;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;

interface IFilter {
    public boolean filter(Integer item);
}

public class Filter {

    private static List select(List list, IFilter filter) {
        List result = new ArrayList();
        for (Iterator it = list.iterator(); it.hasNext();) {
            Integer item = (Integer)it.next();
            if (filter.filter(item)) {
```

```
                result.add(item);
            }
        }
        return result;
    }

    public static void main(String[] args) {
        Integer[] arr = new Integer[]{
                    new Integer(5),
                    new Integer(7),
                    new Integer(13),
                    new Integer(32)
                };
        List orig = Arrays.asList(arr);
        List newList = select(orig,
                    new IFilter() {
                        private Integer threshold
                            = new Integer(10);
                        public boolean filter(Integer item) {
                            return item.compareTo(threshold) > 0;
                        }
                    }
        );
        System.out.println(newList);
    }

}
```

NOTE

Some closure proponents say that the existence of this "named" interface breaks the anonymous concept at the beginning.

First we defined the `IFilter` interface with a `filter()` method that returns a Boolean value indicating whether the condition is satisfied.

Our `Filter` class contains a `select()` method equal to the methods we saw in the earlier Ruby and Python examples. It accepts a list to be handled and the implementation of the `IFilter` interface that filters the values we want in our new list. At the end, we implement the `IFilter` interface as the anonymous inner class in the `select()` method call.

As a result, the program prints this result list to the screen:

```
[13, 32]
```

From this example, we can see even though a similar concept is possible in system-programming languages, the syntax is much more complex. This is an important difference because the natural syntax for some functionality leads to its frequent use, in practice. Closures have simple syntax for passing the

code around the application. That is why you see closures used more often in languages that naturally support them than you see similar structures in other languages (anonymous inner classes in Java, for example).

Hopefully, closures will be added in Java SE 7, which will move Java one step closer to the flexibility of scripting languages.

Summary

In this section of the chapter, I discussed some basic functional characteristics of scripting languages. Many experts tend to categorize a language as scripting or system programming, not by these functional characteristics but by the programming style and the role the language plays in the system. However, these two categorizations are not independent, so to understand how scripting can fit into your development process, it is important to know the functional characteristics of the scripting language and the implications of its design. The differences between system-programming and scripting languages are described later in this chapter, helping us to understand how these two approaches can work together to create systems that feature the strengths of both programming styles.

It is important to note that the characteristics we've discussed thus far are not independent among each other. For example, whether to use static or dynamic typing depends on when the type checking is done. It is hard to implement dynamic typing in a strictly compiled environment. Thus, interpreter and dynamic typing somehow fit naturally together and are usually employed in scripting environments. The same is true for the compiler and static typing found in system-programming environments.

The similar is true for the generation and execution of other programs, which is a natural thing to do in interpreted environments and is not very easy (and thus is rarely done) in compiled environments.

To summarize, these characteristics are usually found in scripting programming environments. Not all languages support all the features described earlier, which is a decision driven by

the primary domain for which the language is used. For example, although Python is a dynamic language, it introduces strong typing, making it more resistible to type misuse and more convenient for development of larger applications.

These characteristics should serve only as a marker when exploring certain languages and their possible use in your development process. More important is the language's programming style, a topic we discuss shortly.

Scripting Languages and Virtual Machines

A recent trend in programming language design is the presence of a virtual machine as one of the vital elements of programming platforms. One of the main elements of the Java Runtime Environment (JRE) is the virtual machine that interprets bytecode and serves as a layer between the application and operating systems. A virtual machine serves as a layer between the application and operating systems in Microsoft's .NET platform as well.

Let's now summarize briefly how the JRE works. Java programs contained in `java` extension source files are compiled to bytecode (files with a `class` extension). As I said earlier, the purpose of bytecode is to provide a compact format for intermediate code and support for platform independence. The JVM is a virtual processor, and like all other processors, it interprets code—bytecode in this case. This is a short description of the JRE, but it is needed for our further discussion. You can find a more comprehensive description at the beginning of Chapter 3, "Scripting Languages Inside the JVM."

Following this, we can say Java is a hybrid compiled-interpreted language. But even with this model, Java cannot be characterized as a scripting language because it lacks all the other features mentioned earlier.

At this point, you are probably asking what this discussion has to do with scripting languages. The point is many modern scripting languages follow the same hybrid concept. Although programs are distributed in script form and are interpreted at

runtime, the things going on in the background are pretty much the same.

Let's look at Python, for example. The Python interpreter consists of a compiler that compiles source code to the intermediate bytecode, and the Python Virtual Machine (PVM) that interprets this code. This process is being done in the background, leaving the impression that the pure Python source code has been interpreted. If the Python interpreter has write privileges on the host system, it caches the generated bytecode in files with a pyc extension (the py extension is used for the scripts or source code). If that script had not been modified since its previous execution, the compilation process would be skipped and the virtual machine could start interpreting the bytecode at once. This could greatly improve the Python script's startup speed. Even if the Python interpreter has no write privileges on the system and the bytecode was not written in files, this compilation process would still be performed. In this case, the bytecode would be kept in memory.

NOTE

Python programs can be distributed in bytecode format, keeping the source code out of the production environment.

From this discussion, we can conclude virtual machines are one of the standard parts of modern scripting languages. So our original dilemma remains. Should we use languages that enforce a certain programming paradigm, and if so, how do we use them? The dynamic and weak typing, closures, complex built-in data structures, and so on, could be implemented in a runtime environment with the virtual machine.

There is nothing to restrict the use of a dynamic (scripting) language on the virtual machines designed for languages such as Java and C#. As long as we implement the compiler appropriate for the target virtual machine's intermediate bytecode, we will receive all the features of the scripting language in this environment. Doing this, we could benefit from the strengths of both the system-programming approach of Java, and the scripting programming model in our software development process.

We focus on projects that bring scripting languages closer to the Java platform later in this book. Also, we discuss where it's appropriate to apply the scripting style of development with traditional Java programming. Before we cover these topics, though, let's take a look at how scripting and system programming compare.

A Comparison of Scripting and System Programming

Every decision made during the language design process is directly related to the programming style used in that language and its usability in the development process.

In this section, I do not intend to imply one style is better than the other is. Instead, my objective is to summarize the strengths and weaknesses of both approaches so that we can proceed to Chapter 2, where I discuss how best to incorporate them into the development process.

Runtime Performance

It is clear programs written in system-programming languages have better runtime performance than equivalent scripts in most cases, for a few reasons:

- The most obvious reason is the runtime presence of the interpreter in scripting languages. Source code analysis and transformation during runtime introduces additional overhead in terms of program execution.
- Another factor influencing runtime performance is typing. Because system-programming languages force strong static typing, machine code created by the compiler is more compact and optimized for the target machine.

The fact that the script could be compiled to intermediate bytecode makes these interpreter performance penalties more acceptable. But the machine code is definitely more optimized than the intermediate code.

We have to take another point of view when talking about runtime performance, however. Many people approach runtime performance by asking which solution is faster. The more important question, which is often neglected, is whether a particular solution is fast enough.

You must take into consideration the tradeoffs between the benefits and the runtime performance that each approach

provides when you are thinking about applying a certain technology in your project. If the solution brings quality to your development process and still is fast enough, you should consider using it.

A recent development trend supports this point of view. Many experts state you should not analyze performance without comparing it to measurements and goals. This leads to debate concerning whether to perform premature or prudent optimization. The latter approach assumes you have a flexible system, and only after you've conducted the performance tests and found the system bottlenecks should you optimize those parts of your code.

Deciding whether scripting is suitable for some tasks in your development process must be driven by the same question. For instance, say you need to load a large amount of data from a file, and developing a system-programming solution to accomplish the task would take twice as long as developing a scripting approach. If both the system-programming and scripting solutions need 1 second to load the data and the interpreter required an additional 0.1 second to compile the script to the bytecode, you should consider scripting to be a fast enough solution for this task. As we see in a moment, scripts are much faster to write (because of the higher level of abstraction they introduce), and the end users of your project probably wouldn't even notice the performance advantage of the system-programming solution that took twice as much time to develop.

If we take another point of view, we can conclude the startup cost of executing programs written in dynamic languages could be close to their compiled alternatives. The first important thing to note is the fact that bytecode is usually smaller than its equivalent machine code. Experts who support this point of view stress that processors have increased in speed much faster than disks have. This leads to the thinking that the in-memory operations of the just-in-time compilers (compiling the bytecode to the machine code) are not much more expensive than the operation of loading the large sequence of machine code from the disk into memory.

To summarize, it is clear system-programming languages are faster than scripting languages. But if you don't need to be

restricted by only one programming language, you should ask yourself another question: What is the best tool for this task? If the development speed is more important and the runtime performance of the scripting solution is acceptable, there is your answer.

Development Speed

I already mentioned dynamic languages lead to faster development processes. A few facts support this assertion.

For one, a statement in a system-programming language executes about five machine instructions. However, a statement in a scripting language executes hundreds or even thousands of instructions. Certainly, this increase is partially due to the presence of the interpreter, but more important is the fact that primitive operations in scripting languages have greater functionality. For example, operations for matching certain patterns in text with regular expressions are as easy to perform as multiplying two integers.

These more powerful statements and built-in data structures lead to a higher level of abstraction that language can provide, as well as much shorter code.

Of course, dynamic typing plays an important role here too. The need to define each variable explicitly with its type requires a lot of typing, and this is time consuming from a developer's perspective. This higher level of abstraction and dynamic typing allows developers to spend more time writing the actual business logic of the application than dealing with the language issues.

Another thing speeding up the scripting development process is the lack of a compile (and linking) phase. Compilation of large programs could be time consuming. Every change in a program written in a system-programming language requires a new compile/link process, which could slow down development a great deal. In scripting, on the other hand, immediately after the code is written or changed, it can be executed (interpreted), leaving more time for the developer to actually write the code.

As you can see, all the things that increase runtime performance, such as compilation and static typing, tend to slow

down development and increase the amount of time needed to build the solution. That is why you hear scripting languages are more human oriented than machine oriented (which isn't the case with system-programming languages).

To emphasize this point further, here is a snippet from David Ascher's article titled "Dynamic Languages—ready for the next challenges, by design" (www.activestate.com/Company/NewsRoom/whitepapers_ADL.plex), which reflects the paradigm of scripting language design:

The driving forces for the creation of each major dynamic language centered on making tasks easier for people, with raw computer performance a secondary concern. As the language implementations have matured, they have enabled programmers to build very efficient software, but that was never their primary focus. Getting the job done fast is typically prioritized above getting the job done so that it runs faster. This approach makes sense when one considers that many programs are run only periodically, and take effectively no time to execute, but can take days, weeks, or months to write. When considering networked applications, where network latency or database accesses tend to be the bottlenecks, the folly of hyper-optimizing the execution time of the wrong parts of the program is even clearer. A notable consequence of this difference in priority is seen in the different types of competition among languages. While system languages compete like CPU manufacturers on performance measured by numeric benchmarks such as LINPACK, dynamic languages compete, less formally, on productivity arguments and, through an indirect measure of productivity, on how "fun" a language is. It is apparently widely believed that fun languages correspond to more productive programmers—a hypothesis that would be interesting to test.

Robustness

Many proponents of the system-programming approach say dynamic typing introduces more bugs in programs because there is no type checking at compile time. From this point of view, it is always good to detect programming errors as soon as

possible. This is certainly true, but as we discuss in a moment, static typing introduces some drawbacks, and programs written in dynamically typed languages could be as solid as programs written in purely statically typed environments. This way of thinking leads to the theory that dynamically typed languages are good for building prototypes quickly, but they are not robust enough for industrial-strength systems.

On the other side stand proponents of dynamic typing. From that point of view, type errors are just one source of bugs in an application, and programs free of type-error problems are not guaranteed to be free of bugs. Their attitude is static typing leads to code much longer and much harder to maintain. Also, static typing requires the developer to spend more of his time and energy working around the limitations of that kind of typing.

Another implication we can glean from this is the importance of testing. Because a successful compilation does not guarantee your program will behave correctly, appropriate testing must be done in both environments. Or as best-selling Java author Bruce Eckel wrote in his book *Thinking in Java* (Prentice Hall):

If it's not tested, it's broken.

Because dynamic typing allows you to implement functionality faster, more time remains for testing. Those fine-grained tests could include testing program behavior for type misuse.

Despite all the hype about type checking, type errors are not common in practice, and they are discovered quickly in the development process. Look at the most obvious example. With no types declared for method parameters, you could easily find yourself calling a method with the wrong order of parameters. But these kinds of errors are obvious and are detected immediately the next time the script is executed. It is highly unlikely this kind of error would make it to distribution if it was tested appropriately.

Another extreme point of view says even statically typed languages are not typed. To clarify this statement, look at the following Java code:

```
List list = new ArrayList();
list.add(new String("Hello"));
list.add(new Integer(77));

Iterator it = list.iterator();
while (it.hasNext()) {
    String item = (String)it.next();
}
```

This code snippet would be compiled with no errors, but at execution time, it would throw a `java.lang.ClassCastException`. This is a classic example of a runtime type error. So what is the problem?

The problem is objects lose their type information when they are going through more-generic structures. In Java, all objects in the container are of type `java.lang.Object`, and they must be converted to the appropriate type (class) as soon as they are released from the container. This is when inappropriate object casting could result in runtime type errors. Because many objects in the application are actually contained in a more-generic structure, this is not an irrelevant issue.

Of course, there is a workaround for this problem in statically typed languages. One solution recently introduced in Java is called *generics*. With generics, you would write the preceding example as follows:

```
List list<String> = new ArrayList<String>();
list.add(new String("Hello"));
list.add(new Integer(77));

Iterator<String> it = list.iterator();
while (it.hasNext()) {
    String item = it.next();
}
```

This way, you are telling the compiler only `String` objects can be placed in this container. An attempt to add an `Integer` object would result in a compilation error. This is a solution to this problem, but like all workarounds, it is not a natural approach.

The fact that scripting programs are smaller and more readable by humans makes them more suitable for code review by a

development team, which is one more way to ensure your application is correct. Guido van Rossum, the creator of the Python language, supported this view when he was asked in an interview whether he would fly an airplane controlled by software written in Python (www.artima.com/intv/strongweakP.html):

You'll never get all the bugs out. Making the code easier to read and write, and more transparent to the team of human readers who will review the source code, may be much more valuable than the narrow-focused type checking that some other compiler offers. There have been reported anecdotes about spacecraft or aircraft crashing because of type-related software bugs, where the compilers weren't enough to save you from the problems.

This discussion is intended just to emphasize one thing: Type errors are just one kind of bug in a program. Early type checking is a good thing, but it is certainly not enough, so conducting appropriate quality assurance procedures (including unit testing) is the only way to build stable and robust systems.

Many huge projects written purely in Python prove the fact that modern scripting languages are ready for building large and stable applications.

Maintenance

A few aspects of scripting make programs written in scripting languages easier to maintain.

The first important aspect is the fact that programs written in scripting languages are shorter than their system-programming equivalents, due to the natural integration of complex data types, more powerful statements, and dynamic typing. Simple logic dictates it is easier to debug and add additional features to a shorter program than to a longer one, regardless of what programming language it was written in. Here's a more descriptive discussion on this topic, taken from the aforementioned Guido van Rossum interview (www.artima.com/intv/speed.html):

This is all very informal, but I heard someone say a good programmer can reasonably maintain about 20,000 lines of code.

Whether that is 20,000 lines of assembler, C, or some high-level language doesn't matter. It's still 20,000 lines. If your language requires fewer lines to express the same ideas, you can spend more time on stuff that otherwise would go beyond those 20,000 lines.

A 20,000-line Python program would probably be a 100,000-line Java or C++ program. It might be a 200,000-line C program, because C offers you even less structure. Looking for a bug or making a systematic change is much more work in a 100,000-line program than in a 20,000-line program. For smaller scales, it works in the same way. A 500-line program feels much different than a 10,000-line program.

The counterargument to this is the claim that static typing also represents a kind of code documentation. Having every variable, method argument, and return result in a defined type makes code more readable. Although this is a valid claim when it comes to method and property declarations, it certainly is not important to document every temporary variable. Also, in almost every programming language you can find a mechanism and tools used to document your code. For example, Java developers usually use the Javadoc tool (http://java.sun.com/j2se/javadoc/) to generate HTML documentation from specially formatted comments in source code. This kind of documentation is more comprehensive and could be used both in scripting and in system-programming languages.

Also, almost every dynamically typed language *permits* explicit type declaration but does not *force* it. Every scripting developer is free to choose where explicit type declarations should be used and where they are sufficient. This could result in both a rapid development environment and readable, documented code.

Extreme Programming

In the past few years, many organizations adopted extreme programming as their software development methodology. The two basic principles of extreme programming are *test-driven development* (TDD) and *refactoring.*

You can view the TDD technique as a kind of revolution in the way people create programs. Instead of performing the following:

1. Write the code.
2. Test it if appropriate.

The TDD cycle incorporates these steps:

1. Write the test for certain program functionality.
2. Write enough code to get it to fail (API).
3. Run the test and watch it fail.
4. Write the whole functionality.
5. Run the code and watch all tests pass.

On top of this development cycle, the extreme programming methodology introduces refactoring as a technique for code improvement and maintenance. Refactoring is the technique of restructuring the existing code body without changing its external behavior. The idea of refactoring is to keep the code design clean, avoid code duplication, and improve bad design. These changes should be small because that way, it is likely we will not break the existing functionality.

After code refactoring, we have to run all the tests again to make sure the program is still behaving according to its design.

I already stated tests are one way to improve our programs' robustness and to prevent type errors in dynamically typed languages. From the refactoring point of view, interpreted languages offer benefits because they skip the compilation process during development. For applications developed using the system-programming language, after every small change (refactoring), you have to do compilation and run tests. Both of these operations could be time consuming on a large code base, so the fact that compilation could be omitted means we can save some time.

Dynamic typing is a real advance in terms of refactoring. Usually, because of laziness or a lack of the big picture, a developer defines a method with as narrow an argument type as he needs at that moment. To reuse that method later, we have to

change the argument type to some more general or complex structure. If this type is a concrete type or does not share the same interface as the one we used previously, we are in trouble. Not only do we have to change that method definition, but also the types of all variables passed to that method as the particular argument. In dynamically typed languages, this problem does not exist. All you need to do is change the method to handle this more general type.

We could amortize these problems in system programming environments with good refactoring tools, which exist for most IDEs today. Again, the real benefit is speed of development. Because scripting languages enable developers to write code faster, they have more time to do appropriate unit testing and to write stub classes. A higher level of abstraction and a dynamic nature make scripted programs more convenient to change, so we can say they naturally fit the extreme programming methodology.

The Hybrid Approach

As we learned earlier in this chapter, neither system-programming nor scripting languages are ideal tools for all development tasks. System-programming languages have good runtime performance, but developing certain functionality and being able to modify that functionality later takes time. Scripting languages, on the other hand, are the opposite. Their flexible and dynamic nature makes them an excellent development environment, but at the cost of runtime performance.

So the real question is not whether you should use a certain system-programming or scripting language for all your development tasks, but where and how each approach fits into your project. Considering today's diverse array of programming platforms and the many ways in which you can integrate them, there is no excuse for a programmer to be stuck with only one programming language. Knowing at least two languages could help you have a better perspective of the task at hand, and the appropriate tool for that task.

You can find a more illustrative description of this principle in Bill Venners's article, "The Best Tool for the Job" (www.artima.com/commentary/langtool.html):

To me, attempting to use one language for every programming task is like attempting to use one tool for every carpentry task. You may really like screwdrivers, and your screwdriver may work great for a job like inserting screws into wood. But what if you're handed a nail? You could conceivably use the butt of the screwdriver's handle and pound that nail into the wood. The trouble is, a) you are likely to put an eye out, and b) you won't be as productive pounding in that nail with a screwdriver as you would with a hammer.

Because learning a new programming language requires so much time and effort, most programmers find it impractical to learn many languages well. But I think most programmers could learn two *languages well. If you program primarily in a systems language, find a scripting language that suits you and learn it well enough to use it regularly. I have found that having both a systems and a scripting language in the toolbox is a powerful combination. You can apply the most appropriate tool to the programming job at hand.*

So if we agree system-programming and scripting languages should be used together for different tasks in project development, two more questions arise. The first, and the most important one, is what tasks are suitable for a certain tool.

The second question concerns what additional characteristics scripting languages should have to fit these development roles.

Let's try to answer these two questions by elaborating on the most common roles (and characteristics) scripting languages had in the past. This gives us a clear vision of how we can apply them to the development challenges in Java projects today, which is the topic of later chapters.

A Case for Scripting

To end our discussion of this topic, I quote John K. Ousterhout, the creator of the Tcl scripting language. In one of his articles (www.tcl.tk/doc/scripting.html), he wrote the following words:

In deciding whether to use a scripting language or a system programming language for a particular task, consider the following questions:

Is the application's main task to connect together pre-existing components?

Will the application manipulate a variety of different kinds of things?

Does the application include a graphical user interface?

Does the application do a lot of string manipulation?

Will the application's functions evolve rapidly over time?

Does the application need to be extensible?

"Yes" answers to these questions suggest that a scripting language will work well for the application. On the other hand, "yes" answers to the following questions suggest that an application is better suited to a system programming language:

Does the application implement complex algorithms or data structures?

Does the application manipulate large datasets (e.g., all the pixels in an image) so that execution speed is critical?

Are the application's functions well-defined and changing slowly?

You could translate Ousterhout's comments as follows: Dynamic languages are well suited for implementing application parts not defined clearly at the time of development, for wiring (gluing) existing components in a loosely coupled manner, and for implementing all those parts that have to be flexible and changeable over time. System languages, on the other hand, are

a good fit for implementing complex algorithms and data structures, and for all those components that are well defined and probably won't be modified extensively in the future.

Conclusion

In this chapter, I explained what scripting languages are and discussed some basic features found in such environments. After that, I compared those features to system-programming languages in some key development areas. Next, I expressed the need for software developers to master at least one representative of both system-programming and scripting languages. And finally, I briefly described suitable tasks for both of these approaches.

Before we proceed to particular technologies that enable usage of scripting languages in Java applications, we focus in more detail on the traditional roles of scripting languages. This is the topic of Chapter 2, and it helps us to better understand scripting and how it can be useful in the overall system infrastructure.

Chapter 2

Appropriate Applications for Scripting Languages

Now that we know the basic characteristics of scripting languages, as well as the advantages and drawbacks of their use in the software development process, we can proceed to a discussion of their use for particular development tasks. We discuss some common scripting language use cases and their traditional roles in development.

This discussion finishes our theoretical introduction to scripting, which is meant to give us a better perspective of how Java projects can benefit from scripting languages. Starting with Chapter 3, "Scripting Languages Inside the JVM," we can fully focus on particular projects and solutions related to the Java platform and Java projects.

Let's now cover some examples of common scripting deployment in an information technology infrastructure.

Wiring

Component-based development (CBD) is usually considered a natural evolution of object-oriented programming. In this scenario, software components can be treated as black boxes that provide services to other components. They also require services from other components. An example of a component-based architecture for Java is the Enterprise JavaBeans (EJB) specification.

People often confuse component-based development with object-oriented technology (OOT) because both of them advocate a separation of the abstraction's interface and its implementation in the software design. Although this paradigm is common for both of these approaches, a few differences are worth noting. In object-oriented programming, this separation is encouraged, but solutions developed with OOP languages can ignore this separation. Component-based development goes a step further, enforcing this separation. So, with this approach, it is guaranteed all components will have defined interfaces for communication with other components found in the system. Another difference is CBD tends to be programming language neutral.

UNIX programming philosophy advocates UNIX programs be written as "components." Do not confuse this with component-based development. We discuss Unix programming philosophy and the importance of scripting in such environments in more detail later in this chapter. For now, let's clarify why UNIX programs can be considered a kind of component.

UNIX programs are written to be small, to do one thing well, and to be easily connected with other programs. They use text streams as a general-purpose interface. Doug McIlroy, the inventor of UNIX pipes, which is described in the following section, and one of the founders of the UNIX tradition, summarized UNIX philosophy as follows:

This is the UNIX philosophy:

Write programs that do one thing and do it well.

Write programs to work together.

Write programs to handle text streams, because that is a universal interface.

Usually, people abbreviate this principle as follows:

Do one thing, do it well.

So from McIlroy's point of view, many UNIX programs can be treated as components. They are usually called *filter programs* (or just *filters*).

What is the role of scripting in such an environment? Scripting languages are often referred to as *glue languages*. Many experts treat them as languages best suited for manipulation and wiring of preexisting components, where the best tools for building these components are system-programming languages. It does not matter whether these components are modules built in an object-oriented language (such as Java), fully featured components (EJBs, for example), libraries of functions written in C, or even UNIX programs. Because scripting languages tend to be dynamically typed, it is easier to use various components and handle them according to the application's need.

A variety of scripting languages are commonly used for component gluing purposes. Among the more popular are UNIX shell languages, Perl, and Tcl.

UNIX Shell Languages

The first UNIX shell (sh) was written in the 1970s, but many variations (csh, zsh, bash, and so on) have emerged since then. The primary purpose of a shell is to let the user type interactive commands, but it also can be used to create new applications from existing applications. One of the crucial prerequisites for this kind of software development was the existence of a specific programming culture on the UNIX platforms. UNIX developers were focused on creating simple filter programs instead of creating complex monolithic applications.

In UNIX, you use pipes to redirect the output from one program to the input of another program. You mark pipes with the | symbol. To demonstrate how to use a shell to glue existing applications together, let's look at the following example:

```
cat test.txt | grep java | wc -l
```

In this simple shell command, we used three UNIX filter programs:

- `cat`, which reads files (or standard input devices) as its input and prints them on its output
- `grep`, which reads its input and prints only lines matching a certain text pattern
- `wc`, which reads its input and prints a number of lines as its output

The previous command does the following:

- Reads the content of the `test.txt` file (`cat test.txt`)
- Redirects that content to the `grep` command, filtering only lines containing the word `java`
- Passes the filtered content to the `wc` program, which prints the number of lines containing the word `java` to standard output (the console)

For the `test.txt` file with the following content, the example shell script prints `2` on execution (because the word `java` appears on two different lines):

```
java is a popular programming
language.
java is
object-oriented
```

This example showed us how a shell could be used to create entirely new functionalities by combining the existing ones, and how easily it can be done.

Perl

Perl is a programming language created by Larry Wall in the late 1980s to enhance the UNIX shell and combine the features of many UNIX scripting tools. It is often referred to as “shell on steroids.”

Perl introduced some new concepts, such as:

- **Powerful data types**—These include dynamic arrays and hashes, among other things.
- **Modularity**—Perl introduced the concept of modules, and today a large community of developers is dedicated to writing open source modules to solve different domain problems. This collection of modules and documentation is known as CPAN (Comprehensive Perl Archive Network). You can find more information on CPAN at www.cpan.org/.
- **Built-in functionalities, such as regular expressions**—These functionalities help developers to cope with everyday programming tasks. For example, regular expressions are powerful tools when it comes to text manipulation.
- **Platform independence**—Unlike shell scripts tied to UNIX, Perl scripts can also be evaluated on other platforms.

Perl is widely adopted in the system administration domain for automating tasks and (as a shell) for development of more complex applications using simple filter programs. Perl also has been used on the Internet to replace C when writing CGI scripts. You can find more details on using Perl for these tasks in the following sections.

Tcl

The *Tool Command Language* (Tcl) is a small language interpreter designed to be easily embedded and extended. Tcl has a simple syntax, where every line represents a command in the following format:

```
command argument1 argument2 . . .
```

For example, in the following line:

```
expr 15 + 17
```

`expr` is the command, and `15`, `17`, and `+` are considered arguments.

Tcl features such important control structures as `if` and `while` as well as a way to define procedures. It is a string-oriented language, meaning a string is the only simple data type available. Languages with this characteristic are usually called *typeless programming languages*. In addition to the string data type, two data structures are available in Tcl: lists and associative arrays (maps). This is basically all you have to know to start writing Tcl scripts, and that is its main strength: simplicity.

The real advantage of Tcl is its architecture, which allows it to be easily extended and embedded into system-programming environments. Commands, which are an essential part of this language, can be written in C or C++. This allows you to use Tcl as a command language that drives components written in lower-level programming languages.

We cover other interesting uses of this language in the following sections.

Prototyping

Development organizations take different approaches to the software development process. One often-used development process is the *prototyping model*. This process consists of a finite number of cycles that include the following steps:

- **Requirements collection and analysis**—This is the initial task common in every development process. Its primary goal is to understand what functionalities the end user needs and how the application fits in its working environment. The essential task of this step is to define the problem the software has to solve.
- **Prototyping**—After requirements are collected and analyzed, the mock-up (prototype) of the project is created.

The prototype is the system capturing essential functionalities of the target system.

- **User evaluation**—The prototype is then returned to the users for evaluation. The users are often not capable of expressing all the functionalities they need at first, and as the prototype is being evaluated, new requirements could arise. The cycle repeats after this step, leading to prototypes of desired functionalities that are more complete.

The prototypes in this process have a few important roles:

- They help find the requirements early.
- By evaluating these prototypes, users complete their knowledge of software requirements, which helps them to better understand the final solution.
- These prototypes can be used to train users before the production system is delivered.

Among the key benefits the prototyping model introduces in development are the following:

- Misunderstandings between end users and software developers are exposed early in the process.
- Missing functionalities are detected early in the process.
- Confusing and complex functionality is detected early in the process.
- A "working" system is available early, which could be used for various tasks, from defining system specifications to user training.

Two general approaches to prototyping, both leading to different results, are throwaway prototyping and evolutionary prototyping.

Throwaway prototyping is used to create and validate system requirements. The prototype is used, along with an initial specification for the creation of a definite system specification. After the prototype is finished, it is usually rewritten to improve the architecture (leading to better maintainability) and performance. Figure 2.1 illustrates the throwaway prototyping process.

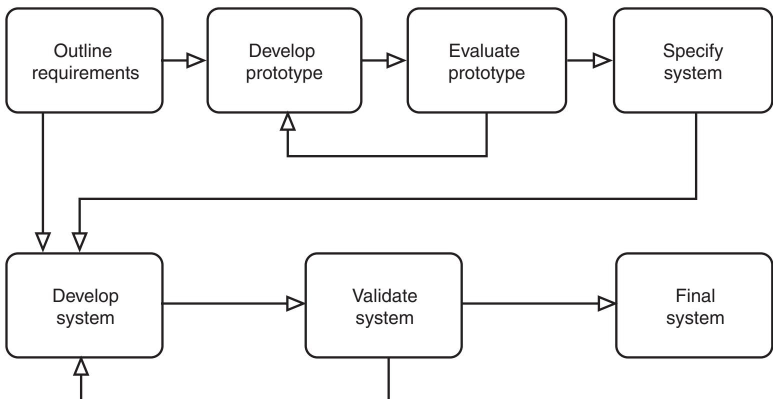


FIGURE 2.1 Throwaway prototyping

In some cases, users are satisfied with the prototype and want to use it as a final system. Although this is not considered a good idea, it is possible to implement all missing features, do performance tuning, and ship the final prototype as the production system. This is called *evolutionary prototyping*, illustrated in Figure 2.2.

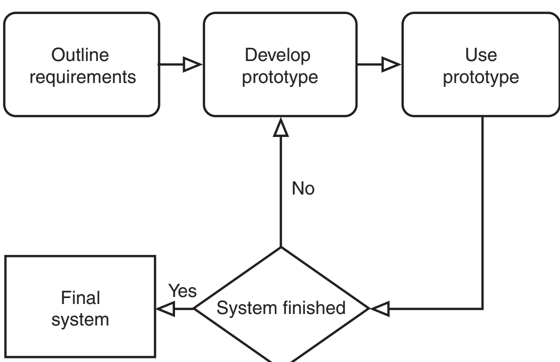


FIGURE 2.2 Evolutionary prototyping

The evolutionary prototyping model, however, has a few weaknesses. First, the system architecture is usually corrupted, which makes further maintenance difficult. The second weakness is a lack of the skills necessary to implement the process in development organizations.

The prototyping model is best suited for use in projects where precise system requirements cannot be specified in advance. For example, if you want to create another word processor or spreadsheet application, you don't really need a prototype. You know what functionalities you want to have, how the user interface should look, and the design patterns to use. But in case you don't know how your final product should look, you can use prototyping to explore that topic further or to help you develop a final solution.

Component-based development suits the prototyping philosophy very well. If you have reusable components, you can build a prototype quickly by gluing existing components together.

High-level dynamic languages are often considered one of the possible tools suitable for prototyping. Because performance is not crucial in this process, rapid development and ease of refactoring are arguments on the scripting side. Scripting languages such as Perl, Python, Tcl, and SmallTalk are often referred to as ideal tools for system prototyping.

Python

Python is a powerful interpreted programming language. In addition to its easy syntax, it provides a mechanism allowing for rapid creation of larger software projects. Python has a dynamic nature—that is, it features dynamic typing—but it is still strongly typed, which is not common to other scripting languages.

Besides built-in, higher-level data structures, Python offers full support for object-oriented programming and modules that enforces code reuse and good architecture in a dynamic and high-level abstraction environment. Python also has mechanisms for exception handling, such as those seen in C++ and Java, which you can use to develop more robust scripts.

Like Tcl, Python also provides an API that allows developers to write modules in languages such as C and C++ and thus extend the language. The Python interpreter could also be embedded into applications written in system-programming languages, and thus provide a programming interface necessary

to extend those applications. These features make Python suitable for system prototyping. After creating a successful prototype, you can rewrite performance-critical modules in a lower-level abstraction language. The final solution would have better overall performance and could be suitable for production, but in the process of its creation, a scripting language as a prototyping tool could be of great help.

Python has been used successfully in many live projects. You can find an example of one such project in Guido van Rossum's interview (www.artima.com/intv/speed.html), a portion of which is reprinted here:

Yahoo Mail started out as a successful Python application. Again, because the developers used Python, they could respond quickly to the user feedback. And that's an application that almost everybody can use. They saw many things wrong with their application, and they responded to that quickly and added new features. Because they were doing something new, they didn't know exactly what people would need from an email Web application. It is different from a program that has your email on your computer. Access times are different. All sorts of things are different. So they were learning about what those differences were. And again, I think Yahoo may now have replaced all the Python code with C++ or some other language, but the Python prototype was essential in order to get there.

I would like to finish this section with a quotation from Frederick P. Brooks, who wrote the following words in his book *The Mythical Man-Month: Essays on Software Engineering*:

The question, therefore, is not whether to build a pilot system and throw it away. You will do that. The question is whether to plan in advance to build a throwaway, or to promise to deliver the throwaway to customers.

This statement emphasizes the importance of the prototyping techniques in modern software development, and as we have seen, scripting languages play an important role in that process.

Customization

Many software projects need to be customizable by end users or developers outside the original development team. Various macros, plug-ins, and extensions are available for many popular applications and platforms. The purpose of these plug-ins and extension points varies. For example, they can help users automate frequently used tasks by combining several lower-level tasks, or they can help change the behavior of applications so that users can customize them to suit their needs.

Various mechanisms are available for providing these plug-ins and extensions in some applications. The most common approach is to expose the public application programming interface (API). This API contains functions, objects, and data structures you can use to create extensions. Next, the application provides mechanisms you can use to register these extensions and map them to certain actions.

In projects that were built in a system-programming language, APIs are generally targeted to the same technology used to develop certain applications (such as C++, for example). The problem with this approach is complexity. End users are rarely experienced programmers, so making them learn the concepts of a complex programming language will not usually give satisfying results.

The idea of a public API could be supported with an embedded language easy to learn and use. Scripting languages are just the right solution for this problem. Because they have a higher level of abstraction, scripting languages are more human-oriented. The concepts of a scripting language are easier to understand than are the details of a lower-level formal language. This makes scripting languages suitable for nonprogrammers and a good tool for customizing software applications.

The specialized languages software vendors use for these tasks are usually called *macro languages*. Macro languages tend to be simple and easy to use, but using general-purpose scripting languages for these tasks instead has its own benefits, as we will see in a moment. Allowing a user to customize different software packages with one general-purpose scripting language can make the customization process even easier.

There are many examples of how scripting languages are used for writing software add-ons and macros. For example, Adobe uses JavaScript as a macro language in many of its products. Visual Basic and JScript (Microsoft's derivation of JavaScript) are widely supported on Windows platforms and applications. In addition, Tcl and Python interpreters can be embedded as libraries in C, C++, and Java applications. This feature, along with the capability to write commands (or modules in Python) using these system languages, makes them a suitable tool for extension, configuration, and management of software applications.

Visual Basic for Applications (VBA)

Microsoft has gone a step further than other software vendors by creating a special dialect of Visual Basic just for programming its Office package, other Microsoft applications, and the applications from other vendors (such as AutoCAD).

This language is called Visual Basic for Applications (VBA). The idea behind VBA is whole applications represented as easily manipulated objects (components). A basic abstraction made in this case is every application is composed of its content and functionality. Content refers to the documents the application is handling, such as spreadsheets in Excel. Functionality represents actions that the application can perform on the content. With VBA, you can easily access any document and work with elements in that document, such as spreadsheet cells.

For example, the following code snippet:

```
Application.Workbooks("test.xls").Worksheets("Orders")
      .Range("C4").Value = 3
```

shows how you can change the cell value in the Excel environment with just one line of code. An additional feature enables users to record their actions in a VBA script. You could use this feature to create stub scripts which could be modified later.

So, with VBA, we have a programming platform made of Windows applications. This is an important fact because by allowing nonprofessional programmers to easily customize their

applications, the whole platform becomes a much more flexible workplace.

Software Development Support

Even if you are using a system-programming language as your main development tool for your projects, you could use scripting languages to automate and speed up tasks that are an essential part of your development process.

In this section, we focus on two longtime examples of such uses of scripting in the software development process.

Project Building

In an environment that uses compiled languages, writing source code is just one step in the software development process. As I already said, developers need to compile the source code and optionally link object files to create executable applications.

If you try to perform this process manually, you will usually end up frustrated and looking for ways to automate it. Everyone who has ever tried to build a Java project is familiar with Ant (ant.apache.org). We discuss Ant later in this section, but now, let's look at another similar tool used long before Ant was created and is still heavily used today. It is the GNU *Make* tool (www.gnu.org/software/make/), which is conceptually the same as Ant, but with a few differences.

The Make tool is best known as a build tool for C programs on Linux, but it can be used for many different tasks. The problem of compiling large programs is you have to execute repeatedly (usually enormously long) lines to invoke the compiler, copy resources, package the distribution, and so on. The first obvious solution is to create a shell script that automates this task, which is certainly a step forward in the right direction. However, it has its problems:

- The free form of such scripts makes them hard for other developers to maintain.
- Large projects take a lot of time to compile, so compiling them from scratch is not time effective.

The Make program enables developers to define *rules* to be used during the build process, and to define dependencies between resources. Rules are defined as a collection of commands to be executed. The syntax of build files also provides basic programming concepts, such as loops and variables. Users with more sophisticated requirements could use shell scripting to achieve the desired behavior of their rules. This way, the first problem is solved, and the script (usually called *Makefile* and located in the source directory or directories) has a well-defined form used for just this purpose. Some experts tend to call this language a *compilation language*. It certainly helps in standardizing the building process and makes maintenance of build scripts easier.

Additionally, these Makefiles could be generated and executed from other scripts because this behavior is a fundamental scripting property, and thus automate this process even more. Other scripting tools are also designed to help developers automate the building process, but they are outside the scope of this book.

The second problem is solved by the *Make* program itself. It compares the timestamps of object files with the timestamps of their appropriate source files. It also checks dependencies among resources, and only sources modified from the last build (and resources depending on that source) are compiled again.

In the early days of Java development, `Make` was used to build Java programs too. Here is a brief Ant history, taken from http://ant.apache.org/faq.html#history, which might clarify this topic a little further:

Initially, Ant was part of the Tomcat code base, when it was donated to the Apache Software Foundation. It was created by James Duncan Davidson, who is also the original author of Tomcat. Ant was there to build Tomcat, nothing else.

Soon thereafter, several open source Java projects realized that Ant could solve the problems they had with Makefiles. Starting with the projects hosted at Jakarta and the old Java Apache project, Ant spread like a virus and is now the build tool of choice for a lot of projects.

Ant uses XML to define targets (rules), and you can extend it using tasks written in Java. Ant introduced many benefits to Java developers; the most important benefit was it is truly platform independent. Even if you could write (to some point) portable Makefiles, that usually doesn't happen, so the user is left changing the script to some extent to successfully build the project.

Even though XML build files are common in the vast majority of Java projects today, many people disagree they are the best solution. Some find their syntax hard to write and read, and some need additional flexibility.

In the current trend of increased scripting popularity for the Java platform, a few solutions exist that synergize scripting and Ant. These solutions could help you write more readable and flexible build files suitable for large and complex projects. We discuss these solutions in Chapter 7, "Practical Scripting in Java."

Testing

We have already discussed the test-driven development principle of extreme programming methodology. Now let's focus on unit testing and what benefits it brings to the software development process.

A *unit test* is a snippet of code written by a developer that proves that certain functionality is behaving as planned in a few common usage scenarios. Unit tests have multiple purposes in the software development process. The most obvious role comes from its definition: to test certain functionality. But although the software is working at this moment, we must be confident our code will still be working properly a year from now, after many modifications would probably have been made on the code base. Andy Hunt and Dave Thomas, the authors of the book *Pragmatic Unit Testing in Java with JUnit*, described this with the following metaphor:

You don't test a bridge by driving a single car over it right down the middle lane on a clear, calm day. That's not sufficient. Similarly, beyond ensuring that the code does what you want, you need to ensure that the code does what you want all of the time,

even when the winds are high, the parameters are suspect, the disk is full, and the network is sluggish.

Unit tests have one more important task, to document our code. We are all familiar with the JavaDocs used to generate documentation of our code using special comment tags. Although this kind of documentation works, it is static, meaning it only shows classes, methods, arguments, and so on, and includes descriptive text to explain what they are used for. You can rarely find examples of how certain methods should be used. Another way is to conduct test cases, which in a certain way show how to use certain objects or methods. From test cases, developers who are new to a project can also learn the boundary values that can be passed and what is critical in certain code libraries.

If we conduct unit testing procedures, we will certainly end up spending a lot of time writing tests for our products. Because we have to spend time testing all the functionalities (or at least the major ones) of the application, we will have less time to contribute to the project itself.

Recent trends show many organizations use scripting languages to implement test cases. If a scripting language for a technology you are using is available and can be embedded easily, you can save a lot of time writing unit tests in that language. As I already said, scripts are shorter and faster to write, so the amount of time needed to write a test case and implement its functionality is going to improve a great deal. A drawback of using scripting languages for this task is test case execution takes longer than with its system-programming equivalent, but still, the overall time and effort involved in unit testing is usually shorter in the first case.

Thus far, we have been discussing unit testing, but it is not the only testing methodology you can use to ensure the quality of the product you'll be delivering to your customers. One interesting testing field is graphical user interface testing. Many tools are available enabling quality assurance personnel to record their actions in a script they can run later to make sure the application (or at least the user interface) is behaving correctly. Scripts can also be generated or modified manually. This customization introduces additional flexibility in the testing process.

Many vendors of such tools have implemented their own domain-specific languages. This usually is not the best idea. As Bret Pettichord, software testing expert and consultant, explains in one of his articles (www.stickyminds.com/sitewide.asp?ObjectId=2326&ObjectType=COL&Function=edetail), many issues could arise from such a decision:

Most test tools come bundled with vendor-specific scripting languages that I call vendorscripts. They are hard to learn, weakly implemented, and most important, they discourage collaboration between testers and developers. Testers deserve full-featured, standardized languages for their test development.

...Because a vendorscript is a specialized language, developers are less likely to know it and little inclined to learn it. I frequently counsel testers and developers to work together on test automation projects. There are many reasons why this is a productive collaboration, but vendorscripts get in the way. They split testers from developers and from each other into tool-specific language isolation. This reduces the opportunities to share, collaborate, and improve the craft.

We have enough trouble bringing developers and testers together in the testing process. The last thing we need is an inherent obstacle from the get-go in our testing tools. Ask a developer to learn Visual Basic? No problem. They can always use that knowledge in the future. Ask a developer to learn a specialized language unique to one tool? You're dreaming. You may already be having trouble getting the developer to pay attention to testing, and now you're asking for more.

Administration and Management

The process of system administration is often related to the process of automation. Common tasks are usually executed periodically and are not easy to do manually. Such tasks could include the deletion of expired users and making various reports using data the system collected during runtime, among others.

The following often-cited system administrators' saying provides the best explanation of this relationship between system administration and automation:

No simple sysadmin task is fun more than twice.

So if you find yourself doing simple manual tasks repeatedly, it is time for you to try automating these tasks.

How quickly these tasks perform is not crucial because they are executed periodically and mostly at times when the system is idle (or at least is not being used heavily). Also, system administration tasks usually require only a small set of steps, so the detailed design does not play an important role.

Scripts are an ideal tool for this kind of task automation. They are easy to write and change, and the performance they provide is generally good for these tasks. As Chad Dickerson, then InfoWorld CTO, wrote in his "Tools for the Short Hike" article (www.infoworld.com/article/03/02/21/08connection_1.html):

When you're going on a one-mile hike, you shouldn't weigh yourself down with a full set of silverware, a saw, scissors, and an inflatable boat just in case you run into a raging river. You take a light backpack, a bottle of water, and a Swiss Army knife.

This thought reflects the essence of small tasks that should be executed only once, or periodically.

A shell can be helpful for writing simple tasks, such as wrapping a few commands, parsing arguments, and thus creating a new command that does just what you need. But for system administration, Perl is known as a master tool. Because of the large number of modules available for use with Perl, Perl administrators can create scripts that are network or database aware. This is far harder to attain with plain shell scripting. This makes Perl an excellent language for remote administration of various hosts, which is usually done in large enterprises and service providers.

For successful automation of system administration tasks, scripting languages are usually used together with some kind of scheduler. The scheduler is a program usually written in a

system-programming language and capable of executing other programs and scripts at precisely specified time intervals. These two components, scripts and schedulers, provide a foundation necessary for successful system administration. It is a good example of practical usage of programs written both in system-programming and scripting languages for system development.

On UNIX systems, you can often find a combination of the *Cron* system scheduler and scripts written in shell, Perl, or Python. They provide a powerful environment for all kinds of administration tasks and are one of the reasons why this platform is popular and widely used.

Even if you think that you, as a Java developer, do not need to care about system administration and that this topic is not relevant, let me reassure you that although system administration is usually mentioned in the context of operating systems, these general practices could be applied to any application (as a system).

For example, say you are developing a highly trafficked Web site in Java. Users could publicly register to the site, but if their accounts are inactive for more than 30 days, they are deleted to free their usernames for other customers. This means every day you have to search for users whose accounts have been idle for a specified amount of time and delete those accounts. This is a classic example of administration tasks found in your Java applications. You can find problems such as these at any step of the application development and maintenance process, from generating heavy reports and compacting data in databases to sending notification e-mails to your clients.

To accomplish this task, would you use a fully featured EJB component, or would you write a five-line script? I would attack it with a script written in a language I could embed into Java and schedule with a Java scheduler. This would save time, could be modified without redeploying the whole system, and would not affect the overall stability of the system.

The rule is simple:

- Do it once manually.
- Write a script in a language of your choice.
- Schedule it if it tends to be a repetitive action.

We cover this topic in detail, and provide concrete examples of a solution for Java systems, in Chapter 7.

User Interface Programming

Development of an application's graphical user interface (GUI) is certainly an important task for a vast majority of applications. The problem with GUI programming, however, is it requires too much work in terms of setting widget properties. Thus, GUI programming is considered a boring task by many software developers. User interfaces are also the target of frequent changes, which makes them difficult to maintain in hard-coded system programming environments.

User interface widgets are components suitable for manipulation by scripting technology. Using scripting for GUI programming leads to faster development and shorter solutions easier to maintain.

The problem with the scripting approach is, as always, runtime performance. The performances are good enough for common user interfaces and applications, but if you are thinking of providing a more graphics-intensive interface, you should probably look for some alternative solution written entirely in a system-programming language.

Tk

Tk is an extension of the Tcl language, which allows GUI programming in an X Windows (a UNIX Windows management system) system. This extension provides Tcl with commands used for manipulation of basic graphical elements needed for building the user interface.

Let's take a look at a simple example:

```
button .hello -text Hello -command {puts stdout "Hello,
World!"}

pack .hello
```

This two-line script shows how easy it is to use Tk to create user interfaces.

The `button` command creates a button named `.hello` (the first argument), with "`Hello`" text on it (the second argument), and specifies the action (print "Hello, World!" on standard output) to be taken when the user presses the button (the third argument). After that, with the `pack` command, we map the button on the screen.

You can use any valid Tcl command in this scenario, so you can create commands that actually drive application logic or use built-in modules and extensions to do various tasks.

Tk is now available as a module for other scripting languages including Perl and Python.

As you can see from this discussion of applications for scripting languages, there is a broad range of use cases where scripting is considered to be an ideal tool for the job.

To complete our theoretical knowledge of scripting languages, we should explore some of the specific development domains where this technology has been proven to be of great help to developers. This is the topic of the following section, and after reading it, you should have a clear picture of where in your projects to use scripting and what benefits and drawbacks to expect.

Use Cases

Now that you understand the typical roles of a scripting language, we briefly discuss specific domains where these technologies have enjoyed remarkable success. Of course, we cannot cover them all in this book, but here are a few examples demonstrating success in the field.

Web Applications

Aiming to make the Web more dynamic, scripting languages have gained popularity again. To understand the role of scripting languages in Web application development, let's go one step back.

As stated on the official W3 Consortium Web site (www.w3.org/), the World Wide Web (WWW) is a "distributed heterogeneous collaborative multimedia information system."

In simpler terms, it is an Internet-based network allowing users from one computer to access information on another computer in a consistent and simple way. For that purpose, the W3 project has defined a few concepts and protocols to make it work.

The first important concept is *hypertext*. Hypertext is text with links to other texts, which allow the user to browse through the documents in an easy and natural way. Each document has a *unified resource locator* (URL) attached to it, which is the networkwide address of the document. The URL's format is one of the fundamental W3 protocols.

The second concept of the WWW is its client-server architecture. To allow consistent and easy access to documents, the user uses a thin client to request a document from the server. This leads to another protocol, called the *HyperText Transfer Protocol* (HTTP), designed to provide a fast, stateless, and extensible way to transfer documents between the server and the client.

The final concept introduced is the *Hypertext Markup Language* (HTML) protocol, which provides a consistent way to structure documents. HTML is the language used for writing hypertext documents. A hypertext document writer defines the document structure, such as the title, body, headings, and so on, and links to other documents by inserting links into them.

The Web community adopted these concepts quickly, and this became one of the crucial reasons for the Internet's popularity.

This architecture was ideal for serving static content, or static documents. The next step was to enable Web servers to serve dynamic content to users. Take a Web site that serves the news to clients as an example. Instead of making new static HTML documents for every bit of news coming into the news desk, people wanted a site that could dynamically create the document with all recent headlines (usually stored in the database). This is how the *dynamic Web* or *Web application* was introduced.

In the early days of Web application development, the Common Gateway Interface (CGI) standard played a crucial role. CGI

is a standard for interfacing external applications with a Web server. When the Web server receives a request for certain documents, it executes the external program, and the program's output is returned to the user. Initially the C programming language was mainly used to write CGI programs. This tended to be a painful thing to do because the program deals with a lot of string manipulation and returns well-structured HTML documents as a result. Because of these facts, many developers tried to find the solution in scripting.

Today, several scripting languages play an important role in Web development. Among the more popular are Perl, PHP, ASP, and JavaScript.

PERL

Because of its advanced text-processing features, Perl has become the de facto language for writing CGI scripts. All the advantages of scripting languages we discussed in Chapter 1, "Introduction to Scripting," were responsible for the wide acceptance of CGI scripts more than CGI programs. The problems attacked in those days were small automation tasks, so there was no need for a more robust infrastructure. Also, the presence of the source code, short code solutions, and flexibility made scripts easier to debug, change, and maintain.

All this made scripting an important topic in the early days of Web application development. Tim O'Reilly, the founder of O'Reilly Media, and Ben Smith wrote the following thoughts in their article, "The Importance of Perl" (http://perl.oreilly.com/news/importance_0498.html), emphasizing the role of Perl in the Internet architecture:

Despite all the press attention to Java and ActiveX, the real job of "activating the Internet" belongs to Perl, a language that is all but invisible to the world of professional technology analysts but looms large in the mind of anyone—webmaster, system administrator, or programmer—whose daily work involves building custom web applications or gluing together programs for purposes their designers had not quite foreseen. As Hassan Schroeder, Sun's first webmaster, remarked: "Perl is the duct tape of the Internet."

As Web applications attracted the interest of a wide developer community, a few weaknesses of the CGI concept needed to be improved upon. Two of the most important weaknesses were:

- **Performance**—CGI was designed to enable Web servers to communicate with external programs. It was not designed for building dynamic Web pages. So every request passed through this interface starts a new process for a program to run. Starting a process on a system consumes both time and resources, so the number of requests the server can handle simultaneously is limited. The same is true if you use a scripting language because a separate interpreter is run for every request.
- **Embedding**—For the same reason, CGI programs (scripts) are not naturally embedded in Web servers. So any useful data structure within a Web server cannot be used in a script. For example, a Web application cannot use a Web server's log to write messages.

A few solutions to the CGI performance problem are available. One popular solution for the *Apache* Web server is *mod_perl*. This module provides a persistent Perl interpreter for Apache Web servers. This approach avoids the runtime penalties of starting up an independent interpreter for every request and enables the creation of Apache modules in Perl.

PHP

The problems with using CGI as a tool for creating Web applications, covered in the section on Perl, opened space for new languages. *PHP* (a recursive acronym for *PHP: Hypertext Preprocessor*) is one such language. Although PHP is referred to as general-purpose scripting language, it gained its popularity as a language for rapid development of Web applications.

Besides the fact it can be embedded in various Web servers, which solves the problems with CGI, its strongest weapon is PHP scripts are embedded in the HTML. Even with Perl, which has advanced string manipulation characteristics, creating HTML output could be a hard task. PHP takes a different

approach; scripts look like ordinary HTML pages, where the PHP code is escaped with a special PHP tag.

Take the following page, for example:

```
<html>
 <head>
  <title>PHP Test</title>
 </head>
 <body>
 <? echo '<p>Hello World</p>'; ?>
</body>
</html>
```

In the preceding example, PHP code is embedded into the page within the <? and ?> markers, which are the start and the end markers of the PHP tag. So what happens when the Web server serves a page like this one?

If the page is saved with a *.php* extension (or some other extension associated with the PHP interpreter), the Web server calls the PHP interpreter before it returns the page. The interpreter processes only the code within the PHP tag and replaces the tag with the result of the code execution (this is done for all tags found within the page). Then the modified page is returned to the client. As such, the preceding example page would look like this:

```
<html>
 <head>
  <title>PHP Test</title>
 </head>
 <body>
 <p>Hello World</p>
</body>
</html>
```

Embedding a scripting language in HTML pages represented a crucial shift in Web application development. From this point on, it was easy to get designers and programmers to work together. Designers can work directly on the layout of HTML documents, while programmers can embed necessary logic in those documents.

Many development organizations adopted this style of Web application development, and PHP became one of the most

popular tools for creating small and midsize Web solutions. However, a few additional factors were crucial to ensure PHP's continued popularity:

- Support for many types of databases.
- Support for a wide range of network protocols, such as IMAP, POP3, SNMP, HTTP, and so on.
- A syntax similar to that of C and Perl.
- Support of object-oriented programming, which would allow better organization of back-end programming and business logic modules. PHP's object model was limited and did not allow developers to follow an object-oriented approach to software development. In PHP 5, this model is improved, opening new possibilities for developers.

You can see PHP is widely adopted, by reading the PHP usage report generated by Netcraft (www.netcraft.com) and accessible at http://www.php.net/usage.php. According to an April 2007 survey, 20,016,421 domains and 1,208,663 IP addresses were hosting PHP.

Another example showing that PHP is ready for large, highly trafficked site development is the fact that Yahoo! decided to replace its proprietary server-side scripting language, called Yscript, with PHP. The main reason was the maintenance cost of its proprietary solution—or, as Yahoo! Engineer, Michael Radwin, said in his often-cited (http://news.com.com/2100-1023-963937.html) PHPCon 2002 conference presentation:

(Yahoo) is a cheap company. (It) can't afford to waste engineering resources.

ASP

Active Server Pages (ASP) is Microsoft's Web development platform, and it has goals similar to those of PHP. ASP uses special tags (<% and %>) to insert scripting snippets into HTML pages. The default language for ASP development is VBScript, but you can use other languages such as JScript.

A comparison of PHP and ASP reveals the following differences:

- **Performance**—PHP is faster than ASP. ASP supports multiple programming languages, and the ASP language compiler runs in a separate process. This makes ASP slower than PHP in terms of handling requests.
- **Platform independence**—ASP runs only on Microsoft Windows-based Web servers, whereas you can deploy PHP on practically any platform. This makes the choice of finding an appropriate Web-hosting provider that suits your needs much harder if you're an ASP developer.
- **Security**—Because many security problems were found for the Internet Information Server (IIS), a Web server primarily used to host ASP applications, there is an attitude that the PHP-Apache environment is more secure.

JavaScript

One more scripting language plays an important role in Web development. Until now, we have talked about server-side problems and solutions, but neither Perl nor PHP is the most widely used scripting language for Web development.

We have already seen some interesting applications of *JavaScript* in software development, but JavaScript found its most important role as a language for client-side scripting in Web applications. We can freely say JavaScript is the most widely used language for this development domain.

Let's go through a short history of JavaScript to see why and how it got this title. In 1995, Netscape added Java support for its Navigator Web browser. The idea was to create a new language making this Java support more accessible to non-Java programmers, and a scripting language was the ideal solution. First, the language was called *LiveScript*, but due to marketing reasons, it soon was renamed JavaScript. This new name would prove to be the source of much confusion in the days to come because it has nothing to do with the Java platform.

Although its primary goal was to control Java applets, JavaScript was adopted as a language for a different purpose. Web developers and designers found it most convenient for making HTML more dynamic.

HTML is a static document, and with Web applications' rising popularity, developers needed a way to make graphical user interfaces out of HTML pages. The missing link was how to handle user events (such as mouse events) on the client side (in the HTML page itself). JavaScript was the ideal solution. Because it could be embedded in HTML pages just as plain text (no bytecode or compilation was needed), it attracted many Web developers.

Another common application is client-side data validation. In an era of low Internet transfer rates, any request to the server was thought of as expensive, so JavaScript was used for form validation in the browser, making it more comfortable for clients. In this way, Web applications were made much more user-friendly and more similar to standard desktop applications.

The next big thing was the introduction of the *Dynamic Hypertext Markup Language* (DHTML), which was designed to make HTML more dynamic and to give developers the ability to manipulate any page element. Its purpose was to enable users to integrate HTML, *Cascading Style Sheets* (CSS), and JavaScript as a universal technology for building user interfaces for Web applications. But then, something went wrong.

The trouble started with the so-called "browsers war." An incompatible proprietary document object model and DHTML made writing cross-browser-compatible software practically impossible. That was the reason why many companies only used plain HTML for their Web user interfaces in those days. It also made space for new technologies for interactive Web pages, such as Macromedia's *Flash*.

These days JavaScript is again in the focus of Web developers. Today, this is because of the *AJAX* (Asynchronous JavaScript And XML) technique used to create more interactive Web applications. The problem with standard Web application development is that display of every page results in a separate HTTP request to the server. You cannot transfer data from the

server to the client browser without reloading the whole page. This is slow for most of the interactive applications, leading to the "static" nature of Web applications. Now, developers can use JavaScript code in their pages to exchange data with the server through `XMLHttpRequest` objects asynchronously. This technique has proved to be ideal for many applications, and thus we witness a large development effort in this area.

All in all, JavaScript, although not its primary goal, is referred to as one of the crucial languages for Web application development.

SUMMARY

As we have seen, in Web application development, developers use scripting languages today more than ever before. But Java is also present in this field of development, and we can see many frameworks and technologies created for easier development of Web solutions on this platform. Because Java is a distributed programming language, it is convenient for the development of large Web solutions. PHP is used much more often in small and medium-size sites because it is easy to learn and enables rapid development.

Where Java and PHP converge is the *Java Specification Request* (JSR) 223 (www.jcp.org/en/jsr/detail?id=223), which aims to develop the general framework for integration of native scripting languages in Java. Because of its popularity, PHP is chosen by the expert group as a reference language for this specification.

This mixing of PHP and Java enables a lot of great things for Web developers, including complete transfer of applications from one language to another, using Java modules from within PHP and vice versa, integrating applications written in these technologies, and using PHP as a general-purpose scripting language in a Java environment. This is important because many Web developers and designers know only PHP, so this framework can be their door to the Java world. Java projects could also benefit from PHP in many areas, one of which is the use of PHP as a template language instead of JSP and Velocity, but I am sure many interesting applications are yet to be found.

Scripting and UNIX

Scripting is at home on the UNIX platform. System administrators, software developers, and users are using it on various occasions, as we have seen in the preceding sections. Some of the major scripting language representatives were born in the UNIX environment, and because they brought many benefits to various user groups, they are widely adopted and used.

In the Linux and open source era, scripting has an important role too. Nikolai Bezroukov wrote the following words in his book *Portraits of Open Source Pioneers* (www.softpanorama.org/People/index.shtml):

I believe that scripting languages represent the most important branch of open source development. First of all the key assumption behind open source is that the source of the program should be readable and modifiable. This is simply not true for a complex 10K lines C or C++ program. So expressiveness of the language and its level are very important factors, that are commonly overlooked. I strongly believe that these languages will be mainstream industrial languages in the coming years.

We can freely say scripting is an important part of the UNIX platform and of the open source movement in general.

Scripting in Games

Scripting languages play an important role in the field of game development, and they have been used in various forms for many years now. Today's game developers are using it to extend their game engines and to enable easier modification of their games. What kind of modifications scripting languages enable depends on the specific game genre and vendor. They vary from scenery generation for adventure games to user interface programming and action handling in action games.

Also, there is no rule regarding which language is found in certain products. Some companies find it most suitable to build a custom solution, and others use general-purpose scripting languages such as Python and Lua in their development process.

SCUMM

The most popular in-house scripting solution for game development is the SCUMM engine, developed at LucasArts.

Back in the 1980s, programmers at LucasArts started to work on a new graphics adventure game called Maniac Mansion. Instead of writing a complicated game engine specific to this game, they decided to write a generic engine for it. They called it SCUMM (Script Creation Utility for Maniac Mansion). The SCUMM engine interprets the bytecode generated by compiling suitable scripts. This concept proved successful, so many games from the same company were released with the same engine.

The use of custom-made scripting languages has its drawbacks, however. First, it is expensive to develop and maintain such a solution. Also, this approach requires your users and developers to learn another programming language just for customizing and extending this application (or this kind of application).

This is why some organizations are using general-purpose scripting languages such as Lua and Python for their production needs.

In conclusion, games are complex to develop, and lessons learned in this industry could be used to create complex software for any other domain. Engines (such as 3D rendering engines), where speed is everything, are implemented in lower-level system-programming languages such as C++. This engine is then exposed to scripts that can easily drive the engine, which presents a new approach to programming specific game features.

Additional Characteristics

Chapter 1 discussed some of the basic characteristics of scripting languages, such as dynamic typing and the existence of an interpreter. These are some fundamental technical issues related to scripting in general. But from all the roles and use cases we covered in this chapter, we can conclude that scripting languages

have a few more characteristics equally important in terms of their successful use. In this section, we cover some of their most important nontechnical characteristics.

Embeddable

Many experts state one of the most important capabilities of a scripting language is *embedding* the language in a variety of environments. An entire solution rarely would be implemented from scratch using a scripting language, so it is necessary to provide a mechanism for evaluating scripts from other environments. Interpreters for some of the languages we covered earlier in the chapter, such as Tcl and Python, come as libraries for system programming languages (such as C). A similar approach is taken with Java, as we will see in the following chapters.

This pertains to general-purpose scripting languages, so what about domain-specific languages such as PHP? Of course, the same is true here. Languages used for development of Web applications, for example, must provide a mechanism enabling them to be embedded into a targeted Web server as well as into HTML documents. This is also valid for languages used in other environments, such as testing tools, applications, and so on.

To conclude, the capability to embed a scripting language is crucial to its successful deployment in certain environments.

Extensible

As I said, many experts refer to scripting languages as the layer of glue that drives components written in a system programming language. To be successful at this task, scripting languages have to provide an interface to components written in that language. Besides this interface, many languages provide the mechanism to write their built-in components in a system programming language. This is usually referred to as being *extensible.*

Extensibility helps us to use those languages in various roles, such as prototyping. Imagine, for example, you prototype with a scripting language and you want to turn that prototype into a shippable product. Often, you can find some modules that

do not have performance good enough for system production. Here is where you can rewrite bottleneck modules to provide better runtime characteristics in the language, with a lower level of abstraction and strong typing, and thus optimize your product.

Easy to Learn and Use

Perhaps this is one of the most important reasons why scripting languages have been widely accepted. Because they have a higher level of abstraction (in other words, because they are much more human-oriented), it is easy to learn the basic concepts of most scripting languages today.

Why is this important? If a language's syntax is simple, professional developers can learn the language in a matter of hours, after which they can develop solutions quickly. This moderate learning curve and rapid development are the main reasons why scripting languages are popular among professional programmers.

But that is not all. Nonprofessional programmers and hobbyists can easily understand the concepts of scripting languages, so they can use them to customize applications or build simple solutions for in-house use. Look, for example, at the Web application domain. Many designers are willing to learn JavaScript or PHP (ASP), so that they can work more closely with developers or make their Web site designs more dynamic.

The simplicity of some scripting languages makes them ideal for teaching basic programming concepts. It is not unusual to see introductory programming courses that cover scripting languages rather than some pseudolanguage. The concepts of system-programming languages such as C, C++, and even Java are too complex to be presented in these courses. Topics such as pointers and memory management are not suitable for these courses, either; explaining the basics of looping, data types, and data structures is easier to do in a higher-level programming language.

There are examples of successful uses of dynamic languages in more advanced programming classes, however. As Ronald Loui, Associate Professor of Computer Science at Washington University in St. Louis, wrote (http://www.cs.wustl.edu/~loui/sigplan):

Most people are surprised when I tell them what language we use in our undergraduate AI programming class. That's understandable. We use GAWK. GAWK, Gnu's version of Aho, Weinberger, and Kernighan's old pattern scanning language, isn't even viewed as a programming language by most people. Like PERL and TCL, most prefer to view it as a "scripting language." It has no objects; it is not functional; it does no built-in logic programming. Their surprise turns to puzzlement when I confide that (a) while the students are allowed to use any language they want; (b) with a single exception, the best work consistently results from those working in GAWK. (Footnote: The exception was a PASCAL programmer who is now an NSF graduate fellow getting a Ph.D. in mathematics at Harvard.) Programmers in C, C++, and LISP haven't even been close (we have not seen work in PROLOG or JAVA).

From all this, we can conclude scripting (higher-level of abstraction) languages have an important role in computer science education.

Conclusion

In this chapter, we discussed some of most popular and widely used scripting languages and their applications over the years. We did this so that we could understand how they fit into the information infrastructure and how Java projects can benefit from them.

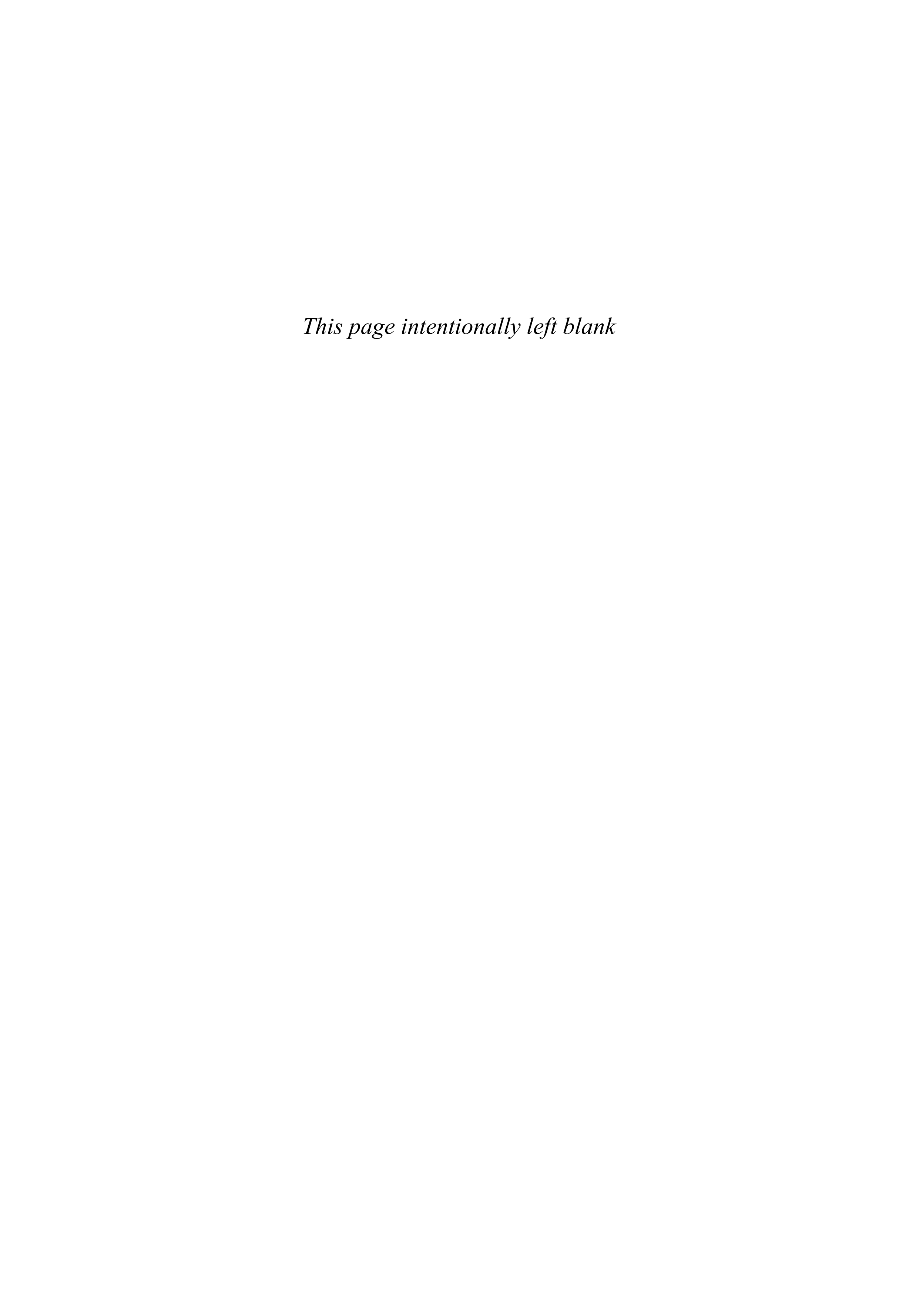
With the increased awareness of scripting in the Java community, many languages we discussed have interpreters written in Java, and some new dynamic languages were created to merge useful features of their ancestors and to adapt the syntax to Java developers.

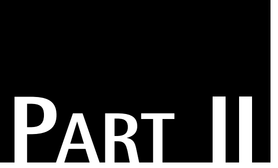
The fact that dynamic languages will be a hot topic in the Java community in the future is almost certain. Even Sun Microsystems, the company mainly responsible for the future of the Java platform, took a few steps in that direction.

In the next three chapters, we discuss some dynamic languages for the Java platform. We do not cover all languages in

this book because this is a wide topic. But we discuss some of the programming concepts already introduced and the mechanisms for embedding them into Java projects.

After that, we focus on how to employ these languages for tasks we described, and explore some patterns for their successful use.


This page intentionally left blank



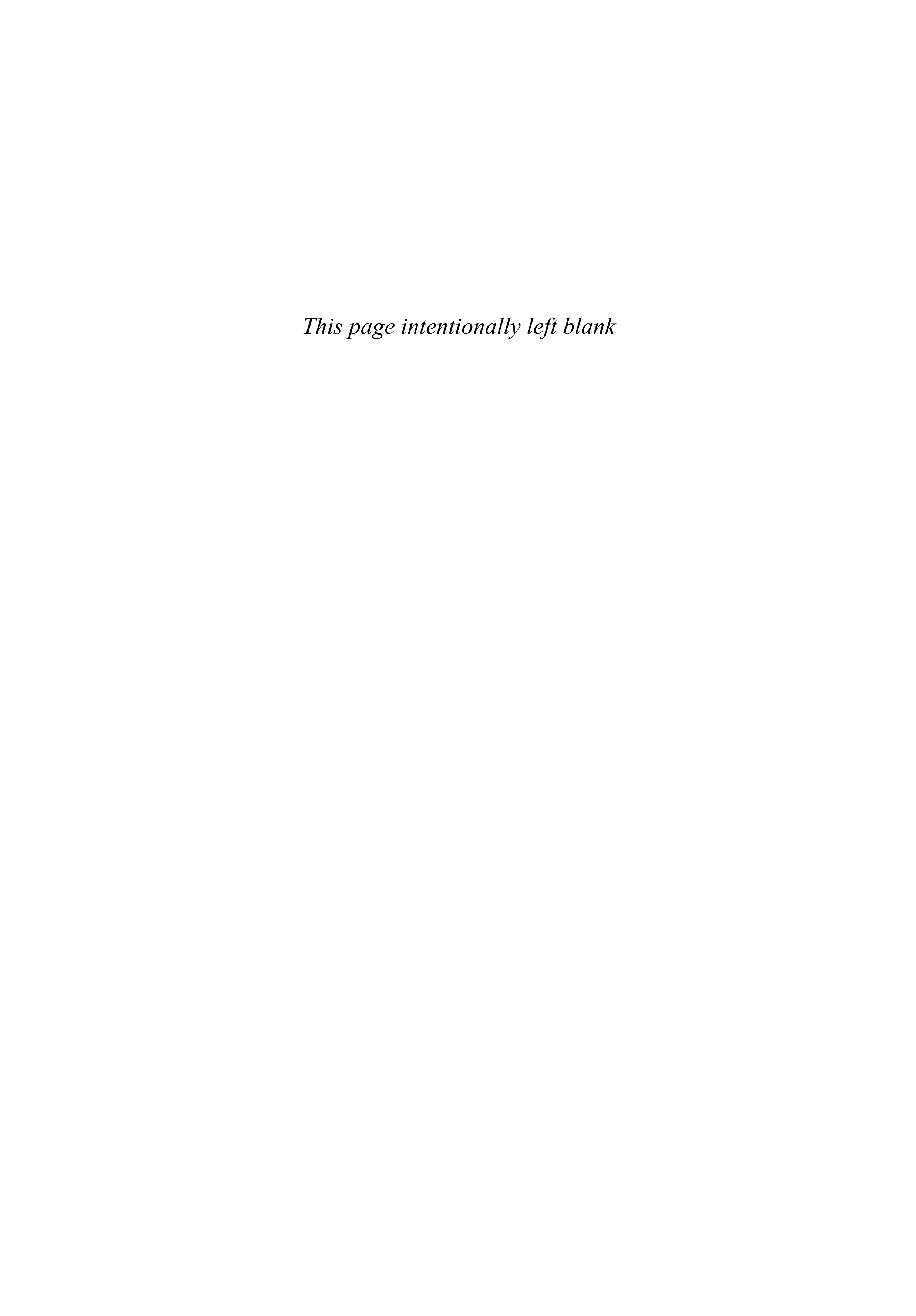
PART II

CHAPTER 3 Scripting Languages Inside the JVM

CHAPTER 4 Groovy

CHAPTER 5 Advanced Groovy Programming

CHAPTER 6 Bean Scripting Framework


This page intentionally left blank

Chapter 3

Scripting Languages Inside the JVM

Since its initial commercial release in 1995, Java has become a widely adopted development platform. It originally was designed as a tool for building client applications that could be easily delivered over a network and run on any platform. But during the dot-com boom, Java won over another industry area: server-side development, Web applications, and enterprise systems.

A few features of the Java language played a vital role in developers' fast adoption of Java. The first feature, of course, was Java's true cross-platform portability. This enabled a unique approach to development, regardless of the platform (or platforms) used in production. A second feature was language support for threading. Java released developers from the burden of having to work with different threading APIs for every operating system available. A third feature was Java's built-in standard implementation of sockets, which enabled development of network-aware applications, making it easier to build distributed systems.

But the real advantage of Java was its simplicity, which eliminated a lot of the pain developers had suffered through when using C++. Java is much simpler than C++. In applications developed with C++, explicit memory management caused many bugs. Explicit memory management gives a lot of power to programmers, but also makes it easy for programmers to crash their systems, especially in large and complex applications. When developing in Java, you don't have to worry about pointers, memory space allocations, and similar issues. The Java Virtual Machine (JVM) does that job for you.

In the same way *implicit memory management* caused Java programs to be less likely to crash because of inappropriate memory handling, implementation of the *garbage collector* in the JVM solved most of the memory-leakage problems. No longer do you need to make explicit destructor calls when an object is no longer needed. The garbage collector (*gc*) deletes unreferenced objects and frees memory in the background, leaving developers to think only about the real business logic of the application.

Java's object-oriented approach was also a big plus. Java has elevated interfaces to first-class status and defined a single inheritance model for classes with the `java.lang.Object` class as a root. Interfaces are collections of methods and constants defining an abstraction. One interface can be implemented several times by different classes, and wherever that interface is expected in the code, any implementation class can be submitted. On the other hand, a class can comprise just one parent class (and implement as many interfaces as it wants). This so-called *single inheritance model*, when powered with interfaces, enables much better designed systems.

Another advantage of Java was dynamic linking. When you compile your Java source files, you don't get one big, executable file with all the necessary libraries linked in. You get one `.class` file for every source (`.java`) file. Those files can be further packed into JAR archives. The class files are not independent, by any means, but they do create an interconnected net. The point is the decision of what class will be used (loaded) in the application is made at runtime. Your Java application

does not have to know exactly which class (that implements a certain interface, for example) will be used at runtime. This makes Java applications extremely modular.

Because a virtual machine represents an additional layer in the overall application architecture, Java applications are usually slower than respective programs written in purely compiled languages, such as C++. But as computers are becoming faster and include more resources (such as memory, for example), the speed disadvantage of Java applications is no longer a key issue. On the other hand, as applications grow larger and more complex, code maintainability and robustness are being stressed. Because of all these features, Java is the dominant development platform in many areas today.

Java's popularity, combined with the open source development trend, led to the large number of Java frameworks, APIs, containers, and so on, existing today. This is exactly the domain of software development that is an excellent fit for a system-programming language such as Java.

The problem in Java development today is it is too hard and complex to glue these components together. When you start planning your future Java project today, start with a list of libraries you want to include. After this, start planning which modules your development team needs to build. Finally, after all the modules are available, put everything together.

The actual business logic that wires modules together, along with the user interface if it is present, is the part of an application changed most often during the development process. Modules' interfaces are rarely changed, but the wiring and behavior of new modules often are changed. For these tasks, where flexibility is crucial, Java is not efficient because it has a strict syntax and does not offer the comfort of scripting languages. Scripts are easy to develop and change, and they require much less code to do this wiring.

Even if Java is the best system-programming language available, there are many places in your system (as we see in Chapter 7, "Practical Scripting in Java," and Chapter 8, "Scripting Patterns") where scripting and Java will work well together. Scripting can make your systems flexible and easy to maintain,

and it can help you do some of the things taking you away from writing actual application code (such as unit testing, debugging, and project building) faster. Scripting also provides you with the ability to do small tasks in a matter of moments.

In this chapter, I walk you through some of the basic Java platform concepts. We see how these concepts enable the Java platform to host different scripting languages. After that, I introduce three popular scripting solutions for the JVM: BeanShell, Jython, and Rhino. Next, I briefly describe some of the most important characteristics of the Groovy programming language. At the end of this chapter, I provide a list of other scripting solutions available for the JVM, along with a brief description.

Under the Hood

Before we start to talk about scripting languages, their features, and applications, it would be good to summarize important information regarding the Java platform and discuss what is really going on under the hood.

The Java architecture consists of four different elements (see Figure 3.1):

- Java programming language
- Java class file format
- Java Virtual Machine (JVM)
- Java application programming interface (API)

The *Java programming language* is what we all refer to as Java. It is a programming language with all the features described earlier.

To execute source code written in Java, a Java compiler has to compile it. This results in *Java class files*. Class files play one of the most important roles in the Java architecture. The format of these files defines a binary form of Java programs, which contains the bytecode executed in the JVM. Java class files are designed to be compact and, most important, platform independent.

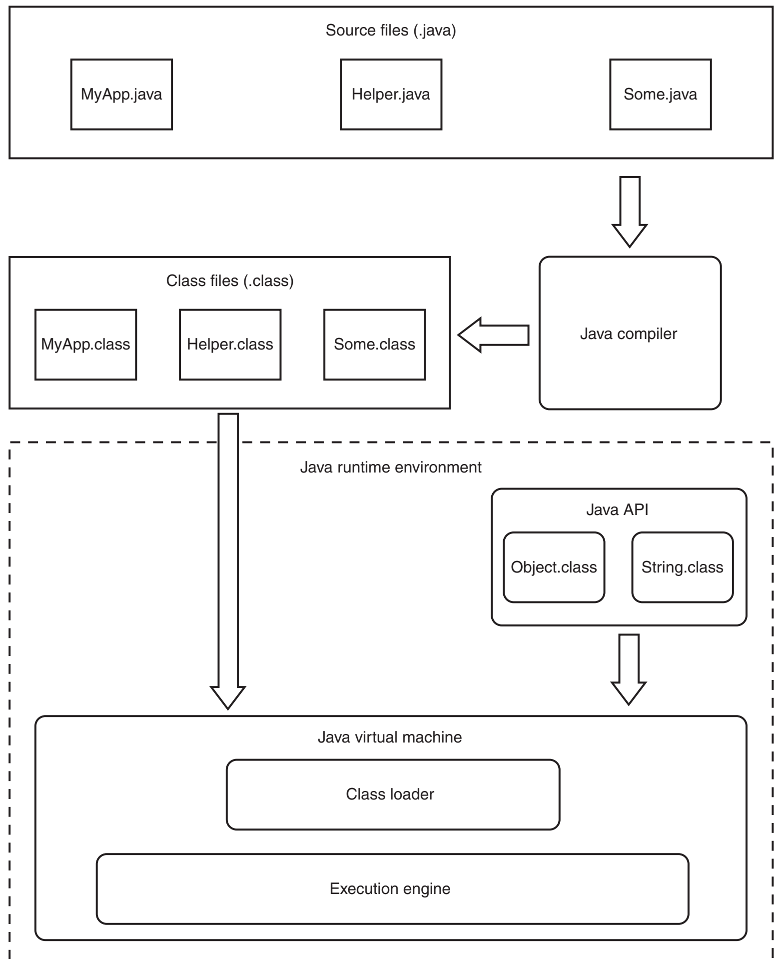


FIGURE 3.1 Java architecture

Classes (located in local class files or somewhere on the network) are then loaded into the JVM for execution. The JVM is an abstract computer whose machine language is the bytecode contained in the class files. Both the bytecode and the JVM are defined by a specification that must be fully implemented by all vendors to have cross-platform compatibility. The JVM has two essential parts: the *class loader* and the *execution engine*.

The *class loader* loads the bytecode from the class files and passes it to the execution engine. There are two types of class loaders: the *bootstrap class loader* and *user-defined class*

loaders. A part of the JVM, the bootstrap class loader is a system default class loader. User-defined class loaders are class loaders we can write ourselves in Java and then instantiate as we can any other Java object. This means we can have multiple class loaders at the same time in our applications. This class-loading mechanism makes the Java platform highly flexible.

The execution engine is the core of the JVM. It is an interpreter of the Java bytecode. The bytecode loaded by the class loader is passed to the execution engine for execution. Usually, the bytecode will be interpreted, but an implementation can also use the just-in-time compiling technique to convert the bytecode to native machine code and then execute that native code.

The last essential part of the Java platform is the *Java API*, which contains classes that implement basic aspects (such as strings and iterators) and allow flexible use of system resources (through files, sockets, and so on). Also, the Java API is designed to be platform agnostic from the point of view of Java developers. For example, if your application has to work with files, you will write the same Java code regardless of whether the application will be run on the UNIX or Windows platform.

The Java API, along with the JVM, forms the *Java runtime environment* (JRE) (as Figure 3.1 shows). At the very least, your host requires the Java runtime environment to run Java applications.

Scripting Language Concepts

From our discussion thus far, we can conclude Java is only one of the programming languages you can use in the JRE. Here are the steps you should take to enable another type of (scripting) language to be used inside the JVM:

1. Implement an interpreter for the specified programming language that runs inside the JVM. The purpose of this interpreter is to parse the language syntax and generate the Java bytecode of parsed scripts.
2. Create a specialized class loader to load parsed scripts. These scripts are loaded as regular Java classes in the JVM. Also, this class loader has to enable scripts to use the Java API and user-defined Java classes.

Another approach people use when working with other languages is to create a compiler that converts source files into Java class files. When you have a valid class file, there is no trace of how it was created. Classes created by a compiler are the same as classes created by compiling Java source files, and thus they can be used in the same way.

These two approaches are not exclusive, so many scripting languages available today allow you to both interpret your scripts and compile them to the Java bytecode. Usually you perform compilation to gain better performance because you will be able to skip the interpreting phase. But the interpreting phase has its place in the development process as well. Its biggest benefit is it enables you to modify scripts easily, without having to compile their source repeatedly.

This is the topic of later chapters. For now, we focus on available scripting alternatives for the Java platform. We briefly cover their features so that you can choose the scripting alternative that best fits your needs. In Chapter 4, "Groovy," and Chapter 5, "Advanced Groovy Programming," we dig into the Groovy scripting language in more detail.

BeanShell

BeanShell (www.beanshell.org) is the first scripting language to introduce the Java syntax. You can think of it as a small, embeddable Java source interpreter. Beyond that, it also extends the Java syntax and introduces some concepts common to scripting languages.

Getting Started

To start playing with BeanShell, you have to complete a few simple steps:

1. Download the latest version of BeanShell from www.beanshell.org. Notice that it is distributed as a single JAR file that is about 250KB in size. This makes it the smallest of all scripting languages available for the JVM because all other solutions that we cover are larger

than 600KB. This is a good thing to know because it could drive your decision regarding which language to embed in your application.

2. Put the downloaded file somewhere in your classpath, and the installation process is complete. For example, on UNIX systems, type something like this in the shell:

```
export CLASSPATH=$CLASSPATH:/path_to_jar_location/
bsh-xx.jar
```

Note that `path_to_jar_location` is an actual directory containing the downloaded JAR, and `xx` in the JAR name is the version of the BeanShell distribution (for example, `/opt/bsh-2.0b4.jar`).

You can run BeanShell in several modes, depending on your intentions. For instance, *interactive mode* is primarily intended for experimentation with scripts and debugging Java applications. This is one of the most popular uses of BeanShell. If you just run the BeanShell JAR with:

```
java -jar bsh-2.0b4.jar
```

you will get a graphical desktop (BeanShell GUI) similar to the one shown in Figure 3.2.

FIGURE 3.2 BeanShell GUI

In this console, you can evaluate your code line by line, as well as open and save scripts to files.

Choosing the Workspace Editor File menu item runs the editor with basic script manipulation features, such as saving the current script or opening an existing script for modification (see Figure 3.3).

> **NOTE**
> If you want to redirect standard input and output streams to this console, select the *Capture System in/out/err* item from the *File* menu.

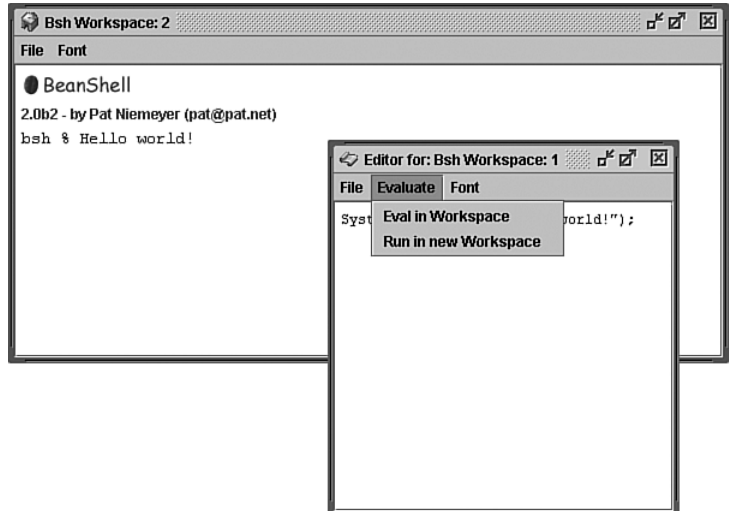


FIGURE 3.3 Workspace editor

This editor is useful if you want to run certain scripts more than once. To evaluate a script, just choose the Eval in Workspace item from the Evaluate menu of the editor.

This BeanShell desktop is not a replacement for a fully featured IDE. It is only a kind of shell with extra features.

Note that implementation of this BeanShell GUI is handled with the `bsh.Console` class. Alternatively, you can start it with:

```
java bsh.Console
```

if the BeanShell is in your classpath.

The `bsh.Interpreter` class implements the BeanShell interpreter. This class contains the `main()` method, so if you want a textual interactive console, run it with:

```
java bsh.Interpreter
```

After this, you can evaluate your statements and exit the interpreter by pressing CTRL-C:

```
BeanShell 2.0b2 - by Pat Niemeyer (pat@pat.net)
bsh % System.out.println("Hello world!");
Hello world!
bsh %
```

If the filename is passed to the interpreter, the script defined in that file will be evaluated:

```
java bsh.Interpreter hello.bsh
```

Optionally, you can pass arguments to that script by appending them to the end of the command line.

Basic Syntax

BeanShell scripts understand Java statements and expressions (methods and classes are discussed in the following sections), which makes them a natural tool for wiring existing Java classes and APIs. For example, look at the following code snippet:

```
Vector vec = new Vector();

for (int i = 0; i < 10; i++) {
    Integer elem = new Integer(i);
    vec.add(elem);
}

Iterator it = vec.iterator();

while(it.hasNext()) {
    System.out.println(it.next());
}
```

As you can see, this looks like standard Java code. The main difference is that you can evaluate this BeanShell script, whereas the equivalent Java class would raise compilation errors. Two crucial factors enable this behavior.

First, BeanShell scripts do not require a class definition and a main() method. Also, some core and extension Java packages

are imported by default, which eliminates many of the import statements. We return to this subject in a later discussion on classpath manipulation in BeanShell, in one of the following sections.

Loosely Typed Syntax

BeanShell shows its “scripting nature” when it comes to the type system. In the earlier example, we declared types for all variables used in the script. BeanShell supports loose typing by allowing you to define variables without an explicit type declaration. The interpreter determines the type at runtime. So, we can rewrite the earlier example in the following manner:

```
vec = new Vector();

for (i = 0; i < 10; i++) {
    elem = new Integer(i);
    vec.add(elem);
}

it = vec.iterator();

while(it.hasNext()) {
    System.out.println(it.next());
}
```

Even if it does not look like a big deal in this simple demo script, loose typing can save you some time during development and can make for a comfortable development environment. Typeless variables can hold any object and primitive value. Variables can even change their type in case they are assigned values of different types.

Loose typing works even in `try/catch` expressions, so you can freely write something like this to catch exceptions:

```
try {
    ...
} catch ( e ) {
    e.printStackTrace();
}
```

Syntax Flavors

Besides loose typing, BeanShell provides a few Java syntax enhancements designed to save you some typing.

JavaBeans

The first enhancement is the way in which you can access JavaBean properties. Now you don't have to use getter and setter methods. BeanShell maps the property name to the adequate method:

```
file = new File("test.txt");
print (file.name); // file.getName()
file.lastModified = new Date().getTime();
// file.setLastModified(new Date().getTime())
print (file.directory); // file.isDirectory()
```

As you noticed in this example, the same syntax stands for `boolean` properties as well. If there is an ambiguity between this mapping and the actual field name, the field name will be used. In these cases, you can use curly braces to enforce use of properties:

```
file = new File("test.txt");
print (file{"name"});
file{"lastModified"} = new Date().getTime();
print (file{"directory"});
```

You also can use this handy syntax with maps and hash tables:

```
map = new HashMap();
map{"title"} = "Java";
print(map{"title"});
```

The `for` Loop

A few Java syntax modifications introduced in Java 1.5 have been present in BeanShell for years. One of those is the modified `for` loop used to iterate over collections and arrays:

```
// Collection

list = new ArrayList();
```

```
list.add("Mike");
list.add("Joe");
list.add("Brus");

for (String item : list) {
    print(item);
}

// Array

users = new String[] {"Mike", "Joe", "Brus"};

for (item : users) {
    print(item);
}

// Iterator

for (String item : list.iterator()) {
    print(item);
}

// String

name = "Dejan";
for (ch : name) {
    print(ch);
}
```

As you can see, besides collections and arrays, this `for` loop can be used on `Iterator` and `String` objects as well.

AUTOBOXING

The term *autoboxing* refers to the automatic conversion of primitive Java types (for example, `int` and `boolean`) to their corresponding Java wrapper classes (for example, `java.lang.Integer` and `java.lang.Boolean`). This feature was first introduced in the J2SE 1.5 (Tiger) platform, and it relieved programmers from much of the tedious work of "boxing" and "unboxing" between wrapper classes and primitive types.

Look at the following Java code, which I wrote prior to the introduction of autoboxing:

```
Integer i = new Integer(12);
int j = 15
if (j > i.intValue())
    System.out.println("j > i");
else
    System.out.println("j > i");
```

With autoboxing, the equivalent Java code looks much more natural:

```
Integer i = 12;
int j = 15
if (j > i)
    System.out.println("j > i");
else
    System.out.println("j > i");
```

BeanShell had this feature a long time before Java did. Take a look at the following example:

```
Vector v = new Vector();
v.put(1);
int x = v.getFirstElement();
```

As you can see, we can use primitive `int` values when we are working with Java collections. Although this has been supported in Java since the 1.5 release, BeanShell allows us to use this feature in older Java runtime environments. Autoboxing can save you a lot of unnecessary typing of type conversion code.

THE `switch-case` STATEMENT

Besides the `for` loop, the `switch-case` statement has been extended to support testing of regular Java objects. The `equals()` method is used to evaluate a condition:

```
file = new File("test.txt");

switch (file.toString()) {
    case "test.csv" : print("Comma separated");
                      break;
    case "test.txt" : print("Plain text");
                      break;
    default         : print("Unknown");
}
```

This kind of `switch` statement eases testing on different values for nonprimitive values and eliminates multiple, hard-to-read `if-else` statements.

Commands

Every scripting language tries to make developers' lives as simple as possible and enable them to achieve more with less code. In that manner, BeanShell defines built-in commands to ease some of the most common development tasks.

You are probably aware of how much time you lose every time you want to display something on standard output:

```
System.out.println(someVariable);
```

The preceding code requires too much typing for such a task.

BeanShell defines the `print()` command for this task, so you can use this in your scripts instead of the `System.out.println()` method:

```
print(someVariable);
```

My intent is just to let you know about this feature, so I will not describe every command that BeanShell offers. You can find a list of commands along with their descriptions in the comprehensive user manual on the BeanShell.org Web site.

Methods

BeanShell allows you to encapsulate often-used code in methods. These methods are not defined inside a class declaration, and as such, they are called *loosely defined* (or *standalone*) methods.

```
double calculateTotal (double subtotal, double tax) {
    return sub * (1 + tax);
}

print(calculateTotal(100, 0.2));

add(first, second) {
    return first + second;
}

print(add(100, 120));
print(add("Demo ", "String"));
```

As you can see in the preceding code, you can define methods without type declarations for arguments and the return value. But you also can enforce types where they are needed. In case you don't define types, the BeanShell interpreter performs type checking and appropriate conversions at runtime.

Objects

BeanShell is designed as a simple language for performing small dynamic tasks. People often use it to write small, unstructured scripts quickly. With that in mind, we can understand why this language does not have full support for Java objects. Instead, it provides a mechanism that is often seen in other scripting languages, called a *method closure*. Methods can be nested, in that you can define a method inside another method. If the method returns a special value called `this`, the result will be treated as an object reference. Look at the following script, for example:

```
invoice() {
    double subtotal;
    double tax;
    double total;

    recalculate() {
        total = subtotal * (1 + tax);
        return total;
    }

    return this;
}

inv = invoice();
inv.subtotal = 100;
inv.tax = 0.2;
print(inv.recalculate());
```

In this code example, we defined the `invoice()` method closure with the `recalculate()` inner method. Note the last statement in the `invoice()` method. It tells us that this is not an ordinary method and that after we call it, we can access its content (variables and methods) the same way we would access objects.

The code after the method definition shows us how to access variables and call methods for this kind of "object."

Implementing Interfaces

Even though BeanShell does not support Java classes, you can still implement Java interfaces with BeanShell. You do this with standard Java anonymous inner classes. A common example of this kind of task is implementation of `ActionListener` interfaces in Swing applications written in BeanShell:

```
buttonHandler = new ActionListener() {
    actionPerformed( event ) {
        print("Thank you");
    }
};

button = new JButton("Click me!");
button.addActionListener( buttonHandler );
frame(button);
```

In this example, we defined the `buttonHandler` variable and assigned an implementation of the `ActionListener` interface to it. This object could be regularly passed around the script wherever this interface is expected.

Another approach that you can take is it to define standalone methods directly in your script and pass `this` as a reference to the script. As such, you could write the previous example like this:

```
actionPerformed( event ) {
    print("Thank you");
}

button = new JButton("Click me!");
button.addActionListener( this );
frame(button);
```

One more interesting thing about working with Java interfaces and BeanShell is that you don't have to implement all the methods defined in the particular interface. Let's demonstrate this with a simple example:

```
package net.scriptinginjava.ch3;

public interface IInvoice {

    public double recalculate();
    public void cancel();

}
```

This is a simple Java interface with two methods defined. Now, let's implement it with the following BeanShell script:

```
import net.scriptinginjava.ch3.IInvoice;

inv = new IInvoice() {
    double recalculate() {
        return 250;
    }
};

print(inv.recalculate());
//inv.cancel();
```

As you can see, we implemented only one of two methods defined in the interface. This is legal, and we can use this method without any implications. But if you uncomment the last line (a call to the unimplemented method), you will get `java.lang.reflect.UndeclaredThrowableException`.

If you want to avoid this behavior, you can implement the `invoke(name, args)` method, which intercepts all calls to unimplemented methods.

Embedding with Java

Probably one of the main reasons that you want to deal with scripting is to enable your Java classes to evaluate some scripts. As we see in Chapter 6, "Bean Scripting Framework," and Chapter 9, "Scripting API," Apache's Bean Scripting Framework (BSF) project and Scripting API (included in JDK 6) are general-purpose frameworks that you can use for this task. But every language that we describe in this chapter has its own mechanism for the same job.

As you can probably guess, the `bsh.Interpreter` class accomplishes this job for BeanShell. Let's make a simple script called `name.bsh`, with the following code:

```
result = "Hello " + name;
System.out.println(result);
name;
```

This script defines the `result` variable and prints it on standard output. Now, we want to write the Java class that will set a value for the `name` variable and evaluate this script:

```
package net.scriptinginjava.ch3;

import java.io.IOException;

import bsh.EvalError;
import bsh.Interpreter;
import bsh.ParseException;
import bsh.TargetError;

public class Name {

    public static void main(String[] args) {
        try {
           Interpreter in = new Interpreter();
           in.set("name", "Dejan");
           String ret =(String)in.source(
                "net/scriptinginjava/ch3/name.bsh"
           );
           System.out.println(ret);
           System.out.println(in.get("result"));
        } catch (ParseException pe) {
           pe.printStackTrace();
        } catch (TargetError te) {
            System.out.println(te.getErrorLineNumber()
               + " " + te.getErrorText());
        } catch (EvalError ee) {
            ee.printStackTrace();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
}
```

The procedure is straightforward:

1. Instantiate the `Interpreter` class.
2. Set mappings for variables using the `set()` method.
3. Call the `source()` method with the name of the script file to evaluate it.

After script evaluation, you can access all variables defined in the script using the `get()` method. Also, the `script()` method returns the value of the last statement of the evaluated script. In this example, we have set the last statement of our script to be the `name` variable, just to demonstrate this behavior.

The earlier code should print the following result on standard output:

```
Hello Dejan
Dejan
Hello Dejan
```

We can learn a few more things from this example. For instance, the `source()` method throws an `EvalError` exception, which indicates an error in script evaluation. This exception defines the following methods:

- `String getErrorText()`
- `int getErrorLineNumber()`
- `String getErrorSourceFile()`

These methods can help you to find where the error was encountered. Also, there are two subclasses of the `EvalError` class:

- **`ParseException`**—Indicates that the particular script could not be parsed
- **`TargetError`**—Indicates that the particular script was parsed correctly and threw an exception during its execution

With these classes in hand, you have a better idea of which error was thrown and why, so you can implement better error handling in your projects.

Besides the `source()` method, you can use the `eval()` method to evaluate scripts. In the case of the `eval()` method, scripts are embedded directly in your Java code. Instead of accepting a filename, this method accepts a `String` variable with the actual BeanShell code that should be executed. The following example is equivalent to the previous Java example, but now we use the `eval()` method instead of `source()`:

```
package net.scriptinginjava.ch3;

import bsh.EvalError;
import bsh.Interpreter;
import bsh.ParseException;
import bsh.TargetError;

public class NameEval {

    public static void main(String[] args) {
        try {
           Interpreter in = new Interpreter();
           in.set("name", "Dejan");
           String ret = (String)in.eval(
                "result = \"Hello \" + name;"
              + "System.out.println(result);"
```

```
            + “name;”
          );
          System.out.println(ret);
          System.out.println(in.get(“result”));
        } catch (ParseException pe) {
          pe.printStackTrace();
        } catch (TargetError te) {
           System.out.println(te.getErrorLineNumber()
             + “ “ + te.getErrorText());
        } catch (EvalError ee) {
           ee.printStackTrace();
        }
    }
}
```

As we said earlier, you can use BeanShell to implement Java interfaces. Then you can load those implementations into Java programs and use them as you would any other regular Java object. To demonstrate this, first we write a simple script that implements our previously defined `IInvoice` interface:

```
double recalculate() {
    return 250;
}

cancel () {
    //Do nothing
}
```

Now we can use the `getInterface()` interpreter method to instantiate this script as an object of a desired interface:

```
package net.scriptinginjava.ch3;

import bsh.Interpreter;

public class InterfaceTest {

    public static void main(String[] args) {
        try {
          Interpreter in = new Interpreter();
          in.source(
            “net/scriptinginjava/ch3/invImplementation.bsh”
          );
          IInvoice inv =
            (IInvoice)in.getInterface(IInvoice.class);
          System.out.println(inv.recalculate());
        } catch (Exception e) {
          e.printStackTrace();
        }
    }
}
```

This by no means covers BeanShell in its entirety. Rather, we have seen some basic principles of programming in this language. Also, we have explored how you can embed it into existing Java code and APIs. With its Java syntax, this language is ideal for Java developers who need a small, modest, and natural scripting solution.

You can find more information on BeanShell in the comprehensive manual on its official site, www.beanshell.org.

Jython

Jython is an implementation of the Python language specification in Java. It leverages Python syntax inside the JVM. Because Python is a programming language specification and Jython is its implementation in Java, I use the term *Python* when I talk about the programming language in general, and the term *Jython* in cases where I talk about specifics of the Jython project.

Python is a high-level, interpreted, object-oriented scripting language. Python has a large community of developers who like its syntax and "feel." They find themselves more productive when developing in Python than with system-programming languages such as Java.

In this section, we do not focus on the Python language. Instead, we describe how Jython works with the Java platform. For a complete reference of the Python language, consult an appropriate book, such as *Learn to Program Using Python* (Addison-Wesley, www.awprofessional.com/title/0201709384).

Getting Started

To start playing with Jython, you need to download it from its official site, www.jython.org. Notice that it is distributed as a single Java class file. This class is an installer that guides you through the installation process (see Figure 3.4).

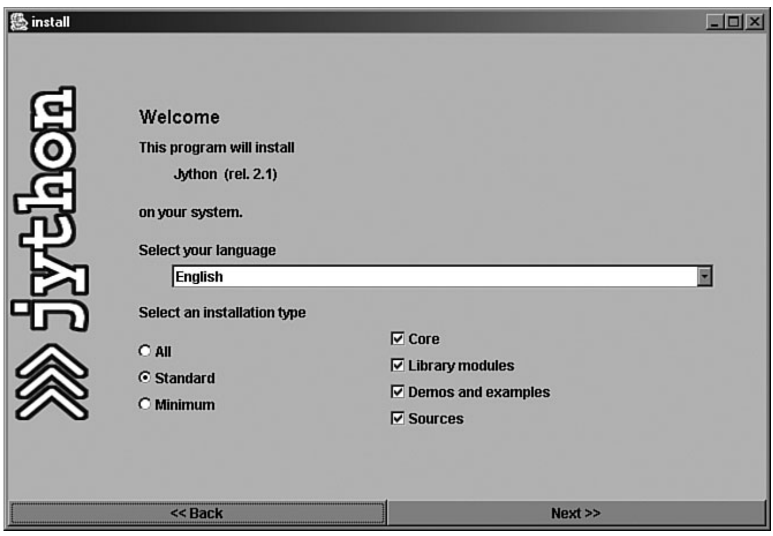


FIGURE 3.4 Jython installer

After the installation process is complete, you will probably want to add Jython to your CLASSPATH and PATH environment variables. On UNIX systems, you need to type something like the following (supposing that Jython is installed in the /opt/jython-2.1 directory):

```
$ export CLASSPATH=$CLASSPATH:/opt/jython-2.1/jython.jar
$ export PATH=$PATH:/opt/jython-2.1
```

You can use Jython in various ways. For instance, you can start it as an interactive interpreter. To do that, just type jython on the command line:

```
Jython 2.1 on java1.4.1 (JIT: null)
Type "copyright", "credits" or "license" for more
information.
>>> print "Hello world"
Hello world
>>>
```

Now, you can evaluate Python statements and expressions line by line. As I already said, this mode is ideal for experimenting with new libraries, debugging, and other similar tasks.

In another mode, Jython works as an interpreter of script files. If you pass the filename to the `jython` command, it will be read and evaluated:

```
$ jython hello.py
```

The interesting thing about this is that you can use Jython to run applications from JAR files. For example:

```
$ jython -jar test.jar
```

searches for the `__run__.py` script in the `test.jar` archive and evaluates it. This could be a useful option if you want to use Jython to start your applications. I discuss this application of scripting languages in more detail in Chapter 7.

As I said, some scripting languages offer the capability to compile scripts to Java class files. Jython is one of them, and you can use the `jythonc` command to achieve this. To demonstrate this, we need a simple "hello world" Python script (`hello.py`):

```
print "Hello world!"
```

Now let's compile it with the following:

```
jythonc hello.py
```

After this compilation process is complete, notice the new `jpywork` directory under the current working directory. This directory contains the results of the compilation process. As you can see, the process is not direct. Instead, the `py` file is translated to the `java` file first, and then the Java source file is compiled with the standard `javac` compiler. Thus, you can find the `hello.java` file in this directory. Table 3.1 contains some of the switches usually used with `jythonc`.

Table 3.1 jythonc switches

Short Version	Long Version	Description
`-p package`	`–package package`	All compiled code is put in the specified package.
`-j jar`	`–jar jar`	Create a JAR archive of the compiled code.
`-d`	`–deep`	Compile all dependencies.
`-c`	`–core`	Include core Jython libraries.
`-a`	`–all`	Include all Jython libraries.
`-A packages`	`–addpackages packages`	Include Java dependencies.
`-w dir`	`–workdir dir`	The (working) output directory for a compiler (./jpywork by default).

Basic Syntax

Let’s now explore values that Python syntax can bring to Java developers. First, dynamic typing is something common to all scripting (dynamic) languages. Python is no exception. Declaration of variables (and their types) before their use is mandatory. Also, they can change their types during program execution, but Python will not automatically cast variable types where it is appropriate. The following are legal calls in Python:

```
a = “Hello world”
a = 2
a += 1.34

print a
```

These calls print `3.34` as a result. But you cannot, for example, mix strings and numbers in this way. If you comment the second line of the preceding script:

```
a = “Hello world”
#a = 2
a += 1.34

print a
```

an exception is raised:

```
Hello world!
Traceback (innermost last):
  File “hello.py”, line 5, in ?
TypeError: __add__ nor __radd__ defined for these operands
```

These examples tell us that Python is a strongly typed language (such as Java) but also that it allows dynamic typing.

Another common characteristic of scripting languages is natural support for some complex data types. In Python, we can find maps and lists more closely integrated with the language:

```
// List example

list = [“mike”, “joe”, “brus”]

print list[1];

// Map example

author = {“firstname” : “Dejan”, “lastname” : “Bosanac”}

print author[“firstname”]
```

Python also provides a more flexible `for` loop to ease iteration over collections:

```
list = [“mike”, “joe”, “brus”]

for item in list :
    print item
```

The next big shift for scripting languages was the paradigm that code and data are interchangeable. In Python, you can freely pass around methods, classes, and modules. Look at the following code snippet, for example:

```
def upper(line) :
    print line.upper()

def someFunc(filename, function) :
    file = open(filename, ‘r’)
    lines = file.readlines(100)
    for line in lines :
        function(line)
    file.close()

someFunc(‘test.txt’, upper)
```

We defined two functions here. As you can see, functions are defined using the `def` keyword followed by the function name, arguments (in brackets), and a colon (:) character. The first function, `upper()`, is a simple function that prints an

uppercase string on standard output. The second function, someFunc(), opens a file with the given name, reads it, and then applies a function (which is passed as its second argument) on each line. As you can see, we passed the upper() function to it, so this script prints the uppercase content of the test.txt file.

Working with Java

As you can see from the basic features covered thus far, Python implements most of the concepts described in Chapter 1, "Introduction to Scripting." Let's now focus on how we can use Python and Java together.

For starters, we want to import some existing Java classes for use in scripts. Python programs are packed into modules, which are similar to Java packages. Thus, we can use Python's import statement to include Java classes and packages. However, there are some differences between the import statements in Python and those in Java.

First, to import all classes from a package, you have to refer to the package name. In Python, you would type the following:

```
import java.lang
```

In Java, you would type the following:

```
import java.lang.*;
```

Another interesting difference is that when you import a package, all subpackages are imported as well.

You can also create an alias to a package to save some time when you refer to classes from that package. For example, after the statement import sun.net.www.http as http, you will refer to the HttpClient class as http.HttpClient instead of as sun.net.www.http.HttpClient.

You can also import only one class of interest from the package by using the from-import statement, as shown here:

```
from java.util import Date
```

Now that we have all the classes we need in the script, we can create objects. When we have imported classes directly with the `from-import` statement, we can use them without referring to their package names:

```
from java.util import Date
now = Date()
print now
```

> **Note**
> Note that in Python no `new` keyword is required for creating objects.

But if we imported the whole package, we must write the whole class name (including the package name):

```
import java.util
now = java.util.Date()
print now
```

Package aliases are handy in these situations and save a lot of typing.

Python also enables import statements to appear anywhere in the script. Look, for example, at the following code snippet:

```
message = "It's now "
from java.util import Date
now = new Date()
print message + now.toString()
```

> **Note**
> We had to call the `toString()` method explicitly to convert the `Date` object to `String`. This is because of Python's strong typing nature.

After we create an object, we are free to call its methods and access its fields just as we would do in Java. However, it's important to note the following issues.

First, problems related to the method overloading mechanism could arise. Python does not support method overloading, so if the Java class has overloaded methods, we can have a problem on our hands. When this situation is encountered, Jython first tries to get the method with the same number of parameters as there are in the call. If there is only one such method, it will be called, and the case will be closed. But in case there is more than one signature of the method with the same number of arguments of different types, Jython will try to find the one that is closest to the arguments in the call. Because this is not guaranteed to be the method that you wanted to call,

you can help Jython by passing arguments with the explicit Java type.

For example, `java.lang.Integer` has the `compareTo()` method, which takes either another `Integer` or any `Object` as a parameter. To be sure that the method with an `Integer` argument is called, we should call it:

```
i.compareTo(Integer(x))
```

where `i` is an Integer object.

Another thing that we should mention about method calls is something we have already seen in BeanShell. Getter and setter methods can be omitted, and properties can be reached directly:

```
from java.io import File
from java.util import Date
file = File("test.txt")
print file.name
print file.directory
```

Implementing Interfaces

Before we start to talk about how to implement Java interfaces with Jython, we must briefly go through Python's class philosophy. Classes are defined similar to loose methods, but with the `class` keyword instead of `def`.

As was the case with standalone methods, class methods are defined using the `def` keyword. The only difference between standalone methods and class methods is that the latter must have at least one argument, usually called `self`, that points to the object itself (it is similar to the `this` keyword in Java). Take a look at the following example:

```
class MyClass :
    i = 5

    def multiply(self, x) :
        return self.i * x

test = MyClass()
print test.multiply(6)
```

We defined a class named MyClass with the multiply() method, which takes one argument (even if it has two arguments in its definition).

You can define the class constructor as a special method called __init__:

```
class MyClass :
    i = 5

    def __init__(self, i) :
        self.i = i

    def multiply(self, x) :
        return self.i * x

test = MyClass(6)
print test.multiply(6)
```

Python supports multiple-inheritance in a limited form. The syntax is:

```
class DerivedClass (BaseClass1, BaseClass2, . . .) :
```

Of course, the single inheritance model is supported as well. It is achieved by specifying just one class, within parentheses. Let's write a simple class derived from the previously defined MyClass:

```
class MyClass :
    i = 5

    def __init__(self, i) :
        self.i = i

    def multiply(self, x) :
        return self.i * x

class Another (MyClass) :
    def add(self, x) :
        return self.i + x

test = Another(6)
print test.add(6)
```

Now that we know the basics of how classes work in Python, we can talk about subclassing Java classes and implementing Java interfaces. It doesn't matter to Jython whether it

is a Python class or a Java class (or interface). Take a look at the following code, for example:

```
from java.awt import *

class MyListener (event.ActionListener):
    def actionPerformed(self, event) :
        print "Thank you"

f = Frame(title="Jython", size=(200,100))
b = Button("Click me")
b.addActionListener(MyListener())
f.add(b)
f.show()
```

Here, we first defined the `MyListener` class that implements the `java.awt.ActionListener` Java interface. After this, we passed an instance of this class to the `addActionListener-Method()` method of the `java.awt.Button` object, just as we would do with an appropriate Java object.

Exception Handling

Exception handling is an important aspect of computer programming. The exception handling mechanism implemented in Python is practically the same as that in Java. This makes it even easier to mix Java and Python code.

This syntax, which you are probably used to in Java:

```
try {
    throw new Exception("Test exception");
} catch (Exception e) {
    System.out.println(e.getMessage());
}
```

is equivalent to the following Python code:

```
try :
    raise Exception, 'Test Exception'
except Exception, e :
    print e
```

Also, there is nothing to stop you from catching Java exceptions in this way. For example, the following is completely valid:

```
from java.io import File, FileReader, IOException
try :
    file = File('test1.txt', 'r')
    reader = FileReader(file)
    line = reader.readln()
except IOException, ioe :
    print ioe
```

And of course, when you are implementing a Java interface in Jython, you can throw (raise) Java exceptions from method implementations.

Embedding with Java

As mentioned earlier, you can compile Python scripts to Java classes by using the `jythonc` tool. In addition, Python scripts can be directly interpreted from Java classes. Some example Java code is:

```
package net.scriptinginjava.ch3.jython;

import org.python.core.PyException;
import org.python.util.PythonInterpreter;

public class Name {

    public static void main(String args[]) {
        try {
            PythonInterpreter in = new PythonInterpreter();
            in.set("name", "Dejan");
            in.execfile(
              "net/scriptinginjava/ch3/jython/name.py"
            );
            System.out.println(in.get("result"));
        } catch (PyException pe) {
            System.out.println(
              pe.value.__tojava__(Exception.class)
            );
        }
    }

}
```

The principle is practically the same as with the BeanShell example. You should create an instance of the `PythonInterpreter` class. Then you can use the `set()` and `get()` methods to do mapping between Java and Python variables. Finally, you can evaluate scripts by using the `execfile()` method.

For the Python script `name.py`, defined as follows:

```
from java.lang import Exception

result = "Hello " + name
print result
#raise Exception, "Bu!"
```

this Java program would print "`Hello Dejan`" twice: once from the script and once from the Java class after the evaluation.

To catch any exception thrown by the script you should test on `org.python.core.PyException`. You can uncomment the last line of the script to simulate this condition.

`PythonInterpreter` is also capable of interpreting Python code in the form of Java strings. For this, you should use the `exec()` or `eval()` method. The following program produces the same result as the one from our previous example (if you use it with an appropriate script):

```
package net.scriptinginjava.ch3.jython;

import org.python.util.PythonInterpreter;


public class NameExec {

    public static void main(String[] args) {
        PythonInterpreter in = new PythonInterpreter();
        in.set("name", "Dejan");
        in.exec("result = \"Hello \" + name");
        in.exec("print result");
        System.out.println(in.get("result"));
    }

}
```

Conclusion

Python is indeed a powerful and beautiful scripting language. But it has more features than we could cover here, and if you don't mind learning another scripting language, Python should be on your to-do list. Then you can use it inside the JVM through the Jython project, just as I have explained here. Of course, for those who want to stick to the Java syntax, there are other scripting solutions, such as BeanShell and Groovy.

Rhino

Back in the 1990s, Netscape was working on the *Javagator* project, a version of *Navigator* written entirely in Java. As part of the project, Netscape developed a Java implementation of the JavaScript interpreter known as *Rhino*. This project outlived the original Javagator project, and Netscape released it to Mozilla.org in 1998.

Most people use JavaScript to enable dynamic client behavior in HTML pages or make asynchronous calls to the Web server (Ajax). However, this language is based on the ECMA-262 Standard (www.ecma-international.org/publications/standards/Ecma-262.htm) and is defined as a "general-purpose cross-platform scripting language." As such, you can use JavaScript for other tasks. Rhino is an exact implementation of JavaScript's core specification, pulled out of the HTML context.

Many Web developers are familiar with JavaScript, so the following information can be a good introduction to Java and Java scripting.

Getting Started

You can download Rhino from its Mozilla project Web page (www.mozilla.org/rhino/). Rhino is located in the `js.jar` file of the distribution, and all you have to do is to put it in your classpath:

```
$ export CLASSPATH=$CLASSPATH:/opt/rhino/js.jar
```

The preceding code is an example of how to install Rhino on a UNIX system if Rhino is extracted in the `/opt/rhino` directory.

The interpreter shell is located in the `org.mozilla.javascript.tools.shell.Main` class, and you can start it with the following command:

```
java org.mozilla.javascript.tools.shell.Main
```

This could be also achieved by executing the `js.jar` file:

```
java -jar /opt/rhino/js.jar
```

When started, the interpreter can be used to evaluate the JavaScript code, as shown in the following code:

```
Rhino 1.5 release 5 2004 03 25
js> print("Hello world!")
Hello world!
js>
```

As is the case with all other languages, this shell is capable of evaluating script files. For that, you should supply the filename as a command-line argument:

```
java org.mozilla.javascript.tools.shell.Main hello.js
```

You also can compile scripts to Java classes by using the `org.mozilla.javascript.tools.jsc.Main` class:

```
$ java org.mozilla.javascript.tools.jsc.Main hello.js
$ java hello
Hello world!
```

Unlike Jython, Rhino does not create an intermediate Java source file. Instead, it directly compiles the script to the appropriate Java class file (`hello.class` in this case).

Working with Java

We do not cover the concepts of the JavaScript language here because it is beyond the scope of this book, and a lot of material already is available on this topic. Instead, we focus on its integration with Java and the Rhino project.

To provide access to Java classes, Rhino defines `Packages`, a top-level variable that contains all packages in the classpath. So, you can reach all classes by referencing their full package names:

```
f = new Packages.java.io.File("test.txt")
```

Only the `java` package is contained as a top-level variable, so instead of the previous statement, you can use the following:

```
f = new java.io.File("test.txt")
```

This can be hard to type, but the `importPackage()` function can simplify things. The purpose of this function is similar to that of Java's `import` statement:

```
importPackage(java.io)
f = new File("test.txt")
```

In this example, we imported the `java.io` package. As a result, we can access classes from this package by using only their names (without a package reference).

Problems could arise if you try to import the `java.lang` package, due to name collisions between Java classes and corresponding JavaScript classes, such as `Object`, `Boolean`, `String`, and so on, so you should avoid importing this package if possible.

After you create an object, you can call its methods and access its fields:

```
importPackage(java.io)
f = new File("test.txt")
print(f.name)
```

Note that in the previous example, `f.name` represents an abbreviation for `f.getName()`. The purpose of this is to simplify working with JavaBeans, as we already discussed in the sections on BeanShell and Jython.

As is the case with Jython, when it comes to calls to overloaded Java methods, Rhino chooses the most appropriate one at runtime, according to the type of the passed arguments.

Implementing Interfaces

Rhino enables developers to implement Java interfaces or subclass Java classes with JavaScript code. Let's now rewrite in JavaScript the examples that we used in the BeanShell and Jython sections.

```
importPackage(java.awt);
importPackage(java.awt.event);

MyListener = {
    actionPerformed : function(event) {
        print("Thank you")
    }
}

f = new Frame("JS frame")
f.setSize(200, 100)
b = new Button("Click me!")
b.addActionListener(new ActionListener(MyListener))
f.add(b)
f.show()
```

This example script implemented a JavaScript class with the `actionPerformed()`method, which accepts one argument. Then we passed an instance of this class to the `addActionListener()` method of the `java.awt.Button` class. Note the syntax of that call; in Java, it is not legal to instantiate interfaces, so this call `(new ActionListener)` triggers a compilation error. In Rhino, however, this is the way to create objects of a desired interface, providing an object that implements that interface as an argument. So, the following statement

```
new ActionListener(MyListener)
```

creates an instance of the `MyListener` class and casts it to the `ActionListener` interface.

We can now elaborate on this example. Because JavaScript functions are first-class language citizens, you can pass them around as method arguments. Rhino additionally allows you to use functions to implement interfaces, in case an interface defines just one method:

```
importPackage(java.awt);
importPackage(java.awt.event);

function doIt() {
    print("Thank you")
}

f = new Frame("JS frame")
f.setSize(200, 100)
b = new Button("Click me!")
b.addActionListener(doIt)
f.add(b)
f.show()
```

In the previous example, we defined the `doIt()`function and passed it to the `addActionListener()` method. This simplifies implementation of simple interfaces a great deal.

JavaAdapter

An alternative approach to instantiating JavaScript classes as Java objects is to use the constructor of the `JavaAdapter` class. Instead of writing:

```
listener = new java.awt.ActionListener(MyListener)
```

you can do something like this:

```
listener = new JavaAdapter(java.awt.ActionListener,
MyListener)
```

This approach is useful when you want your JavaScript class to implement more than one Java interface or inherit some class in addition. For example, this statement:

```
listener = new JavaAdapter(java.awt.ActionListener
                          , java.lang.Runnable
                          , MyListener)
```

creates an object that extends the `java.awt.ActionListener` and `java.lang.Runnable` interfaces.

Embedding with Java

As you might expect, Rhino also enables Java applications to evaluate JavaScript scripts. To demonstrate this functionality, let's first define a simple JavaScript file (`name.js`):

```
result = "Hello " + name
```

This script defines only the `result` variable consisting of a string and the `name` variable that we pass from our Java application. Now take a look at the following Java code evaluating this script:

```
package net.scriptinginjava.ch3.rhino;

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

import org.mozilla.javascript.Context;
import org.mozilla.javascript.JavaScriptException;
import org.mozilla.javascript.Scriptable;


public class Name {

    public static void main(String[] args) {
        try {
            Context cx = Context.enter();
            Scriptable scope = cx.initStandardObjects();
            scope.put("name", scope, "Dejan");
            FileReader script = new FileReader(
               "net/scriptinginjava/ch3/rhino/name.js"
            );
            Object result = cx.evaluateReader(
                scope, script,"<cmd>", 1, null
            );
            System.out.println(result);
            System.out.println(scope.get("result", scope));
        } catch (IOException ioe) {
            ioe.printStackTrace();
        } catch (JavaScriptException jse) {
            jse.printStackTrace();
        } finally {
            Context.exit();
        }
    }
}
```

Before executing the script, a `Context` object must be associated with the current thread. You do this with the `enter()` static method.

The JavaScript language defines three types of objects. The first type is *built-in (standard) objects* such as `String` and `Boolean`. The second type is *host objects* that are specific to a runtime environment (we return to these objects in the next section). The third type is *user-defined* objects. To allow our scripts to use standard JavaScript objects, we have to call the `initStandardObjects()` method. This method returns a `ScriptableObject` instance that you can use to bind variables to scripts. You should use the `get()` and `set()` methods of the `ScriptableObject` class to make these bindings.

Scripts are evaluated by using either the `evaluateReader()` or `evaluateString()` method. You use the `evaluateReader()`

method to evaluate code from the appropriate java.io.Reader object (we used java.io.FileReader in the earlier example). You use the evaluateString() method to evaluate code contained in String variables. If we had used this method in our example, the earlier call would look like this:

```
Object result = cx.evaluateString(
                     Scope
                    ,"result = \"Hello \" + name"
                    ,"<cmd>", 1, null
                );
```

Both of these methods return a value of the last statement in the evaluated script. In our example, it is the value of the result variable.

When this program is executed, the following text is printed on standard output:

```
Hello Dejan
Hello Dejan
```

The first line represents a result of the evaluateReader() (or evaluateString()) function, and the second line represents a value of the result variable obtained from the binding context. Obviously, these two lines are identical.

You should release every context associated with the thread after use. To handle this, we put the exit() method call of the Context class in the finally block.

In addition to these standard embedding operations, Rhino provides some extra features for Java developers. The first feature worth mentioning is the capability to call JavaScript functions from Java code.

Consider the following script, for example:

```
function hello(name) {
    return "Hello " + name
}
```

This script defines a single hello() function. Let's now call it from the Java program:

```
package net.scriptinginjava.ch3.rhino;

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

import org.mozilla.javascript.Context;
import org.mozilla.javascript.Function;
import org.mozilla.javascript.JavaScriptException;
import org.mozilla.javascript.Scriptable;


public class Func {

    public static void main(String[] args) {
        try {
            Context cx = Context.enter();
            Scriptable scope = cx.initStandardObjects();
            FileReader script =
              new FileReader(
                "net/scriptinginjava/ch3/rhino/func.js"
            );
            cx.evaluateReader(scope, script,"<cmd>", 1, null);
            Object func = scope.get("hello", scope);
            if (func instanceof Function) {
                Object funcArgs[] = {"Dejan"};
                Object result = ((Function)func).call(
                    cx, scope, scope, funcArgs
                );
                System.out.println(result);
            }
        } catch (IOException ioe) {
            ioe.printStackTrace();
        } catch (JavaScriptException jse) {
            jse.printStackTrace();
        } finally {
            Context.exit();
        }
    }
}
```

Because functions are first-class language citizens, we can obtain them with the `get()` method, just as we would with any other `Object`. JavaScript functions are instances of the `org.mozilla.javascript.Function` class that defines the `call()` method for their execution. Arguments are passed as an array of objects, and that is all we need to know to call JavaScript functions.

Host Objects

As already discussed, host objects are specific to a certain environment. For example, the `Document` and `Window` objects that

you usually find in HTML pages are host objects. Rhino enables you to implement host objects in Java and use them in your scripts.

To demonstrate this process, we go through an imaginary implementation of the HTML `form` object. We do not implement it for real, but instead, just show the basic principle of how it can be done.

We want to enable Rhino to evaluate scripts like this one:

```
f = new form();
f.method
f.method = "POST"
f.method
f.reset()
f.submit()
```

Because `form` is not a standard JavaScript object, we have to implement it and define it within the runtime environment:

```
package net.scriptinginjava.ch3.rhino;

import org.mozilla.javascript.ScriptableObject;

public class Form extends ScriptableObject {

    private String method = "GET";

    public Form() {

    }

    public String getClassName() {
        return "form";
    }

    public void jsConstructor() {
        System.out.println("Creating form");
    }

    public void jsFunction_submit() {
        System.out.println("Submitting form");
    }

    public void jsFunction_reset() {
        System.out.println("Resetting form");
    }

    public String jsGet_method() {
        System.out.println(method);
        return method;
    }
```

```
    public void jsSet_method(String method) {
        this.method = method;
    }
}
```

Every JavaScript object implementation must implement the `org.mozilla.javascript.Scriptable` interface. The usual approach, however, is to extend the abstract `org.mozilla.javascript.ScriptableObject` class that implements this interface. We have chosen this approach for our example. The only abstract method of this class is `getClassName()`, which returns a name of this class. This name is used to reference the class in JavaScript. Also, we have to define a zero-argument constructor to enable object creation.

Methods whose names start with the `js` prefix represent methods and properties that you can use from scripts. Here are some rules to follow regarding these method names:

- `jsConstructor` represents a JavaScript constructor for this class. Here, we defined a zero-argument constructor, so you could create this object with:
  `f = new form();`
- Methods that start with the `jsFunction_` prefix are method definitions. The `jsFunction_reset()` Java method enables calls to the `reset()` method in JavaScript.
- Object properties are defined with the `jsGet_` and `jsSet_` prefixed methods. The first method enables you to get a value of the specified property, and the second method enables you to set it (in JavaScript, of course).

All we need to do now to finish our example is to bind this host object to the context and call our script:

```
package net.scriptinginjava.ch3.rhino;

import java.io.FileReader;

import org.mozilla.javascript.Context;
import org.mozilla.javascript.Scriptable;
import org.mozilla.javascript.ScriptableObject;

public class FormTest {
```

```
    public static void main(String[] args) {
        try {
            Context cx = Context.enter();
            Scriptable scope = cx.initStandardObjects();
            ScriptableObject.defineClass(scope,
              Form.class);
            FileReader script =
              new FileReader(
                "net/scriptinginjava/ch3/rhino/form.js"
            );
           cx.evaluateReader(scope, script,"<cmd>", 1, null);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            Context.exit();
        }
    }
}
```

This example is almost the same as our introductory embedding example. Note that we used the `defineClass()` method call that defines the `Form` class as the JavaScript host object.

As a result, this program prints the following output:

```
Creating form
GET
POST
Resetting form
Submitting form
```

These lines are printed from the appropriate methods of the `Form` class.

Conclusion

JavaScript is a popular programming language used mostly for manipulation of HTML objects. People are used to its syntax, and a lot of good literature is available, which makes it even easier to learn. These are just some of the reasons you might consider Rhino as a scripting solution for your projects.

Groovy

Groovy (http://groovy.codehaus.org) is a scripting language for the JVM with Java-like syntax. Because we cover it in detail in

the next two chapters, here we describe only the basic benefits that it brings to developers.

Groovy is also a subject of standardization by the Java Community Process (JCP) under the Java Specification Request 241 (www.jcp.org/en/jsr/detail?id=241). This expert group tries to standardize the Groovy language to enable various vendor implementations. Groovy is not targeted for any platform in particular; just as a standard scripting language solution for Java developers.

Groovy uses Java syntax, but on top of that, it provides some concepts that are borrowed from Python, Ruby, and SmallTalk. For starters, it supports weak typing, where variables do not have to be defined before the first use and no type declarations should be made. Next, Groovy compiles directly to the Java bytecode. Unlike Jython, no intermediate `.java` files are made.

Unlike BeanShell, Groovy scripts can implement real Java classes and of course subclass existing ones (or implement interfaces). Also, you can load a script as an interface implementation and use it as a regular Java object in your Java application.

Groovy provides native language support for maps and lists, which is similar to Python's syntax. And as is the case with all the other languages talked about in this chapter, Groovy simplifies JavaBeans by deprecating getter and setter methods. Furthermore, the Java syntax is enhanced with more powerful loops, switch statements, autoboxing, and operator overloading.

Because it is under heavy development, Groovy already offers many extensions that could ease many day-to-day development tasks. You can find extensions that can help you work with servlets, SQL, XML, and so on.

Overall, Groovy provides Java-like syntax empowered with proven scripting concepts, full access to the Java platform, and many extensions, which simplifies programming some of the more advanced tasks. All these features make Groovy a powerful programming environment.

For more details on Groovy, refer to Chapters 4 and 5.

Other Scripting Languages

In addition to the languages we've discussed thus far, many other projects exist that enable some kind of scripting for the JVM. Some of them represent an implementation of existing languages, and others implement completely new concepts. We could not cover all of them in this book, but here are brief descriptions in case you want to try them within the Java context.

JRuby

JRuby (http://jruby.codehaus.org/) is a Java implementation of a Ruby interpreter. Ruby (http://ruby-lang.org) is a pure object-oriented scripting language. It has a simple syntax and is often thought of as a competitor to Perl and Python.

Ruby appeared on the radar of the Web developer community with the *Ruby on Rails* project (www.rubyonrails.com/). It is a powerful framework for the development of Web applications.

Since late 2006, JRuby is officially backed by Sun Microsystems due to Web developers' increased interest in the Ruby on Rails Web framework and huge development possibilities of integrating it into the Java platform. I see it as a growing trend in this field of development and a technology that deserves a special consideration and appropriate covering material.

Tcl/Java

The *Tcl/Java* (http://tcljava.sourceforge.net/) project integrates Java with the Tcl scripting language. It consists of two distinct packages: *Tcl Blend,* which uses JNI to enable Tcl scripts to access Java objects, and *Jacl*, which is a Java implementation of a Tcl interpreter.

JudoScript

JudoScript (www.judoscript.com/judo.html) is a functional scripting language. It uses a JavaScript-like syntax and programming model, but on top of that, it provides domain-specific functional support. This programming philosophy

enables users to specify what they want to do rather than how they want to do it.

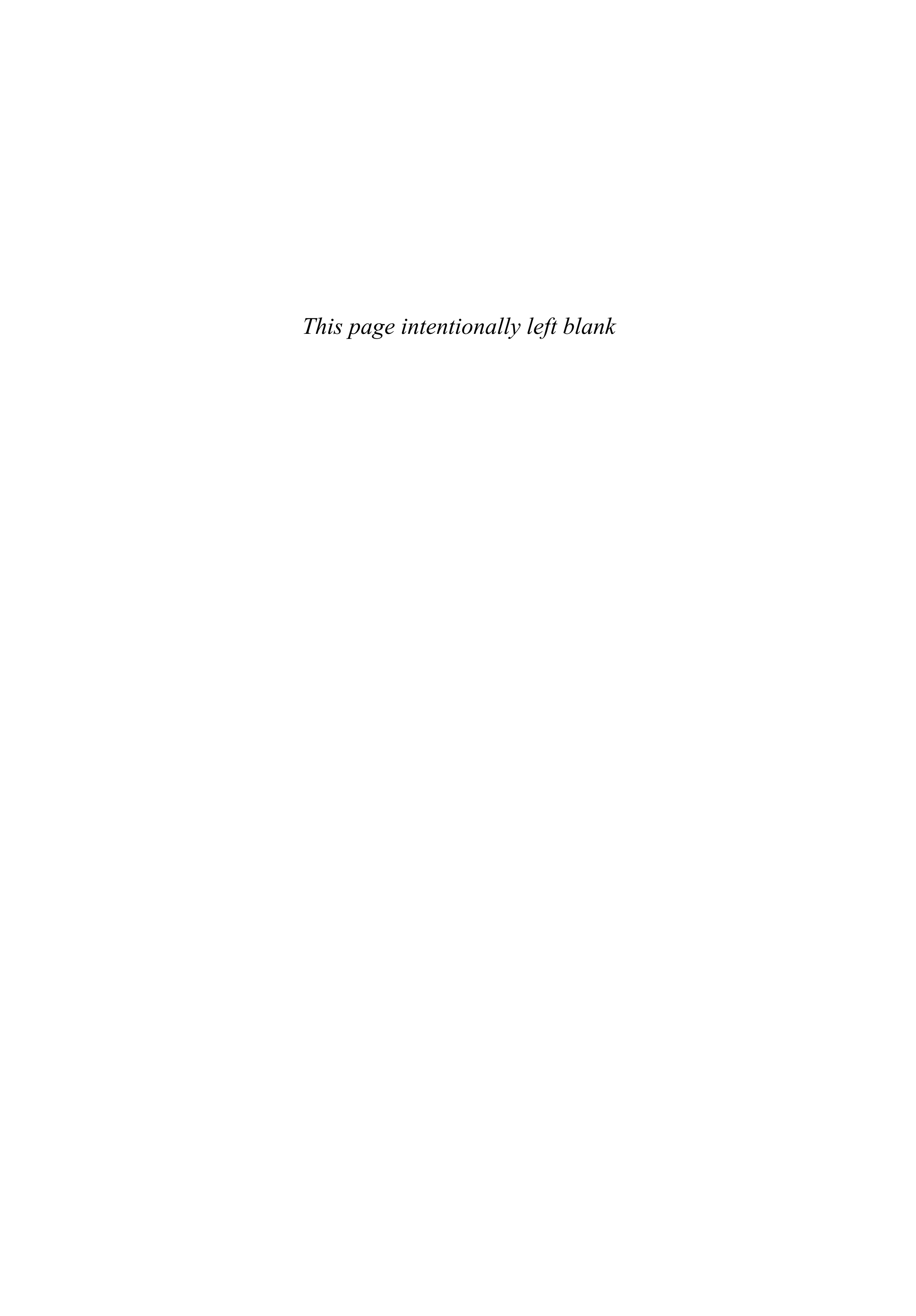
ObjectScript

ObjectScript (http://objectscript.sourceforge.net/) is another Java-like scripting language. It has syntax that is easy to learn and use. It also has some advanced features, such as exception handling and support for threading.

Conclusion

In this chapter, we learned the basic principles behind implementations of scripting languages for the JVM. Also, we covered some of the most popular scripting languages. Although we didn't discuss their syntax and modules, we did focus on how they integrate with the Java platform.

In the following two chapters, we cover Groovy in detail. Chapter 4 contains details about Groovy's syntax, Java integration, and some security-related issues. Chapter 5 focuses on Groovy extensions that you can use to integrate it with other technologies, such as Java servlets, relational databases, XML, and so on.



This page intentionally left blank

CHAPTER 4

GROOVY

In the “Groovy” section of Chapter 3, “Scripting Languages Inside the JVM,” we listed some of the basic concepts that Groovy introduces. In this chapter, we take a closer look at Groovy. This chapter covers the following topics:

- The process of running and compiling Groovy scripts.
- How Groovy implements some of the scripting concepts with Java-like syntax.
- Other benefits that Groovy has to offer developers.
- How Groovy can be integrated with Java. In this section of the chapter, we see how we can evaluate Groovy scripts from Java applications and how to implement Java interfaces in Groovy.
- Security issues related to integration of Groovy and Java.

We cover advanced Groovy programming and describe extensions available for the Groovy language in Chapter 5, “Advanced Groovy Programming.”

Why Groovy?

The questions developers often ask are

- Why do we need another scripting language?
- What benefits does Groovy bring that make it a better solution than Jython or BeanShell?

In this section, I answer these questions, but as always, you should find your own way to compare these technologies and choose the one that fits your development needs best.

Groovy's most important characteristic is its Java-like syntax. In Chapter 1, "Introduction to Scripting," we discussed the need for developers to have a solid knowledge of at least two programming languages, one system-programming language and one scripting language. To use Jython for everyday programming tasks, Java developers should learn Python well. I do not discuss Python syntax here, and I really think it's a great programming language, but Python syntax could be a hard catch for Java developers. This fact could make Java developers more reluctant to learn and try to experiment with Jython, which could result in them rejecting the whole scripting concept altogether. Also, using Java and Jython together everyday requires a lot of mental shifts between these two completely different language syntaxes, which can result in slower development and the introduction of new bugs. So the point is this: If Groovy could implement all the language concepts Jython offers, in syntax close to that used by Java developers, it would make it easier for them to learn and use.

BeanShell, on the other hand, has Java-like syntax but is designed to be a small and natural scripting solution in Java. For example, BeanShell does not enable you to define classes, so the only way to implement the Java interface with it is by using anonymous inner classes. This is fine for some applications, but developers who need a fully featured, object-oriented scripting language can find that in Groovy.

Installation

If you have not installed Groovy on your development platform yet, and you want to run examples while you are reading the material in this chapter, now is a good time to do it.

You can find more information on installing Groovy in Appendix A, "Groovy Installation," or on Groovy's Web site, http://groovy.codehaus.org.

Running Groovy Scripts

You can run Groovy scripts in a number of different ways:

- Using an interactive command-line shell
- Using an interactive console
- Evaluating a script file
- Compiling a Groovy script into a Java class file

I discuss the first three methods in the remainder of this section and cover how to compile a Groovy script into a Java class file in the next section.

Using the Interactive Shell

One way to use Groovy is through the interactive shell. It works like any other shell, and you start it by typing the following on the command line:

```
$ groovysh
```

You enter the script, which is basically the set of statements, line by line, pressing the Enter key at the end of each line. The script is evaluated after you type the `go` or `execute` command, and the result is displayed in the following rows. The prompt for entering statements of the next script is displayed below the result.

After you have finished with your work in the shell, type either the `exit` or the `quit` command to return to your operating system.

Listing 4.1 shows an example of using the Groovy shell. In this listing, we execute two one-line scripts and then exit the shell.

Listing 4.1 Groovy Shell

```
$ groovysh
Let's get Groovy!
================
Version: 1.0-beta-5 JVM: 1.5.0_08-b03
Type 'exit' to terminate the shell
Type 'help' for command help

1> println "Execute me!"
2> go
Execute me!

1> println "Once more!"
2> go
Once more!

1> exit
$
```

> **Note**
> In the rest of this chapter, the result is bolded for the examples that are written in the shell, just as in the preceding example. It should help to make code samples more readable.

In addition to `go`, `execute`, and `exit`, there are a few more useful commands that could help you to successfully finish your tasks:

- **`help`**—Displays the help screen, along with a short description of all the commands available in the shell
- **`discard`**—Discards the current script
- **`display`**—Displays all statements of the current script:

  ```
  1> println "Line one"
  2> println "Line two"
  3> display
  1> println "Line one"
  2> println "Line two"
  3>
  ```

- **`explain`**—Displays the parse tree for the current script

Using the Interactive Console

If you prefer using the graphical user interface, you can also use Groovy's Swing-based console (see Figure 4.1).

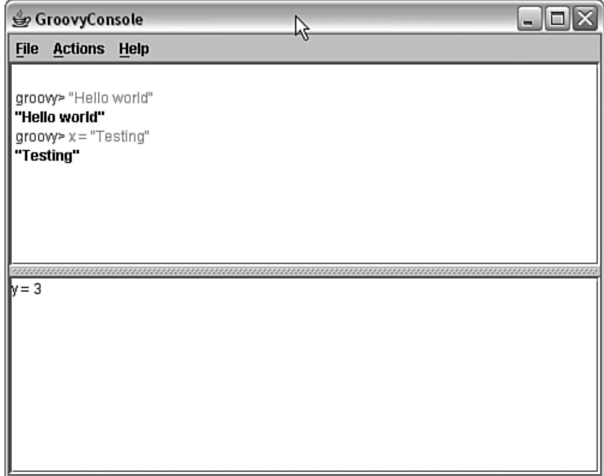


FIGURE 4.1 Groovy interactive console

Although the Groovy interactive console is still pretty modest in terms of functionality, it does offer you the ability to save your script after evaluation or to open the script that you were working on before. To run a console, type the following in the command prompt:

```
$ groovyConsole
```

Evaluating the Script File

As with all other scripting languages, use the shell and console only for the simplest tasks. The usual procedure is to create a file that contains a script so that you can evaluate it more than once.

It is common for Groovy scripts to have a `.groovy` extension, but that is not necessary. After you have created a script file such as this `first.groovy` example file:

```
println “Hello world!”
```

you can execute it by typing:

```
$ groovy first.groovy
```

You can pass any number of parameters to your script by writing them after the script name:

```
$ groovy first.groovy test 123
```

Note

These methods for running Groovy scripts are certainly useful, but most Java programmers are tied to IDEs such as the open source Eclipse (http://www.eclipse.org) project.

It is important to allow Java developers to write, execute, and debug Groovy scripts in their environment of choice. Thus, on the official Groovy site, you can find a Groovy plug-in for some of the most popular IDEs today. You can find more details on how to install Groovy support for every unsupported Java editor (or IDE) in Appendix B, "Groovy IDE Support."

Compiling Groovy Scripts

Another useful Groovy feature is the ability to compile Groovy scripts directly into Java classes. This could be valuable in situations where the Groovy script is used to prototype some features. After the prototype is accepted, the script could be compiled to an equivalent Java class to improve the overall performance of the solution (you can find more information on this topic, along with some concrete examples, in Chapter 8, "Scripting Patterns").

To compile a Groovy script from the command line, just use the `groovyc` compiler (it is similar to the standard `javac` compiler):

```
groovyc first.groovy
```

This creates the `first.class` file in the same directory (you can explicitly change the destination of the generated Java class with the `-d` switch).

The generated Java class has the same name as the original script and preserves its original functionality. The difference is now it can be integrated directly in your Java application (or executed using the `java` command). In this way, we can improve the performance of scripts because there are no runtime penalties of interpreter startup and script evaluation.

Dependencies

Groovy depends on the ASM library (the Java bytecode manipulation framework, http://asm.objectweb.org/). So if you want to execute the class generated by compiling the Groovy script, the appropriate versions of the `groovy.jar` and `asm.jar` files must be in the classpath. These JAR files are located in the `lib/` directory of the Groovy distribution. The command line that executes the earlier `first.class` file could look like this:

```
java -classpath \
$GROOVY_HOME/groovy.jar:$GROOVY_HOME/lib/asm.jar:. \
 first
```

The following shows up on the display as a result of the execution:

```
Hello world!
```

Classpath

In addition to the `groovy` and `asm` JAR files, your scripts usually need access to other Java APIs and libraries. Most commonly you need the JDBC driver (if you want to issue some database queries), but it can be any existing Java library as well. These APIs should be defined somewhere in your classpath, just as they would be for the regular Java applications. There are three ways to set the classpath for Groovy. The first two in the following list are well known from Java, and the third is Groovy specific:

- **`classpath` (or `cp`) switch in the command line**—This approach passes the `-classpath` (or `-cp`) switch to the `groovyc` command.

  ```
  groovyc -cp /home/dejanb/quartz.jar first.groovy
  ```

- **`CLASSPATH` environment variable**—Groovy uses the classpath defined with the `CLASSPATH` environment variable, just like Java does. To set the `CLASSPATH`

variable for UNIX-like systems, use export CLASSPATH=/home/dejanb/quartz.jar. For Windows platforms, use set CLASSPATH=C:\dejanb\quartz.jar.

- **~/groovy/.lib/ directory**—For easier configuration on UNIX-like platforms, you can omit the first two methods and just put all the JAR files used by your scripts in the ~/groovy/.lib/ directory. Groovy looks up these JAR files in that directory and puts them in the classpath for you.

Ant Task

Because the Ant project (http://ant.apache.org/) is the most popular building tool for Java, a <groovyc> Ant task is available that you can use to compile Groovy scripts. Its syntax is similar to that of the <javac> task, commonly used for compilation of Java source files.

Listing 4.2 shows the <groovyc> task definition and usage in the build.xml Ant files.

Listing 4.2 Compiling Groovy Scripts with Ant

```
<project name="groovy project" default="compile">

<path id="lib">
        <fileset dir="/opt/groovy/lib/" />
</path>

<taskdef
    name="groovyc"
    classname="org.codehaus.groovy.ant.Groovyc"
    classpathref="lib"
/>

<target name="compile">
        <groovyc destdir="." srcdir="." listfiles="true">
                <classpath refid="lib"/>
        </groovyc>
</target>
</project>
```

First, we defined the classpath and included all the JAR files from the $GROOVY_HOME/lib directory. Strictly speaking, we could include just the appropriate groovy and asm JAR files,

but by doing it the way we've done it here, we can be sure we have all the libraries we will ever need in the future. You should change the `dir` value in the `<fileset>` tag to the value of the `$GROOVY_HOME` environment variable on your system.

Next, the Groovy task is defined with the `<taskdef>` tag. As you can see, implementation of this task can be found in the `org.codehaus.groovy.ant.Groovyc` class. After these initialization steps are made, the use of the `<groovyc>` task is straightforward. Specify a directory that contains your Groovy scripts (`srcdir`), a directory where the compiled classes will be generated (`destdir`), and a classpath to be used (`classpath`). If you create the `build.xml` file in this way, go to the folder where your `first.groovy` file resides and type the following in the command line (of course, Ant should be properly installed):

```
ant
```

You should get the following message on the console:

```
Buildfile: build.xml

compile:
  [groovyc] Compiling 1 source file to /home/dejanb/scripts
  [groovyc] /home/dejanb/scripts/first.groovy

BUILD SUCCESSFUL
Total time: 3 seconds
```

Afterward, you should find the `first.class` file in the same directory.

Script Structure

Now that we have covered the basics of how to configure Groovy and run the scripts, it's time to dive into the language details. For starters, let's focus on the possible structures that Groovy scripts can have. As I already explained, Groovy's language syntax is close to that of Java and is readable by any Java programmer. So, one way to write the scripts is to encapsulate your code into classes, just as we do with our Java applications.

The first thing that differentiates Groovy scripts from Java source files is statements and methods not associated with any class can be defined in the script. These kinds of methods (and statements) are usually called *standalone* or *loose methods*.

For example, take a look at Listing 4.3 (`structure.groovy`).

Listing 4.3 Groovy Script Structure

```
class Hello {
    public static String hello() {
        return "Hello";
    }

}

class World {
    public static String world() {
        return "world";
    }
}

println Hello.hello() + " " + World.world() +
    termination();

def termination() {
    return "!";
}
```

This script contains two class definitions: one statement definition and one function definition. After running the script, the statement is executed, and the `Hello world!` message is printed on the screen.

The new keyword, `def`, is introduced for defining standalone methods. Also, another thing you can learn from this example is methods (and variables) do not have to be defined before their initial use. As you can see, we used the `termination()` method before it was defined.

It is interesting to see what would happen after compiling the script that contains class definitions and standalone method declarations. First of all, a class with the same name as the script would be generated. This class would contain all the loose code in the script. All loose statements are grouped in the `run()` method, which is invoked from that class's `main()` method. The standalone methods are represented as static

methods of the generated class; thus, they can be invoked from other Java classes in the standard fashion.

Groovy classes are not different from Java classes by any means, so every class would be compiled to the separate file with the `.class` extension. The file would be named after the class.

According to this discussion, we expect three `.class` files to appear (`structure.class`, `hello.class`, and `world.class`) if we compile the `structure.groovy` script.

This discussion has one important implication, however; if you define the class that has the same name as the script, you cannot write loose statements outside that class. For example, if the definition of the `Invoice` class is written in the `Invoice.groovy` file, like this:

```
class Invoice {

    public recalculate() {
        // … do something
    }

}
```

it is compiled without errors, and the `Invoice.class` file is created. But if we try to add some extra code outside the class, like this:

```
class Invoice {

    public recalculate() {
        // … do something
    }

}

inv = new Invoice()
inv.recalculate()
```

a `duplicate class declaration` error arises. The error is caused by name collision because the Groovy compiler tries to generate two `Invoice` classes: one with the loose statements in the script and another for the `Invoice` class itself.

This does not occur often because scripts tend to be either class definitions or executable code in loose statements. However, if it does occur, you can solve the problem simply by collecting the loose statements in the `main()` method of the class, just as in Java:

```
class Invoice {

    public recalculate() {
        // … do something
    }

    static main(args) {
        inv = new Invoice()
        inv.recalculate()
    }

}
```

There is, of course, another possible situation where we have the class definition with the `main()` method, loose statements in the script, and no name collision. If this were the case, loose statements would be executed.

Command-Line Arguments

Proper handling of arguments passed to the program is one of the main programming tasks. The command-line arguments passed to the script are mapped to the `args` variable, and they can be used directly in the script. This variable is equivalent to the argument of the `main()` method in Java classes.

Listing 4.4 shows an example of handling command-line arguments.

Listing 4.4 Handling Command-Line Arguments in Groovy

```
if (args.size() != 2)
    println "Usage: groovy first.groovy arg1 arg2"
else
    println args[0] + " " + args[1]
```

In this example, we first test whether two arguments are present. If they are not present, we print the appropriate message, or print their values otherwise.

Language Syntax

In this section, we dig deeper into Groovy language syntax. We see differences between Java and Groovy syntax and how Groovy implements some of the scripting language concepts in the Java fashion.

Java Compatibility

To begin, it is important to understand that Groovy is not backward compatible with Java. You can call Java code from Groovy and vice versa, but if you try to rename `.java` files to `.groovy`, you will probably have problems evaluating the files.

The most obvious example of this incompatibility is the lack of the standard `for` loop implementation in Groovy. Let's take the following Java code (`wrong.java`) as an example:

```
class Wrong {

    public static void main(String[] args) {
        for (int i=0; i < 10; i++) {
            System.out.println(i);
        }
    }

}
```

This code is regularly compiled and executed in Java, but if you try to make a Groovy script out of it:

```
$ cp wrong.java wrong.groovy
$ groovy wrong.groovy
```

you will get an exception.

The important point to take away from this discussion is that you should not expect existing Java code to be valid Groovy code. Total backward compatibility with Java is the long-term Groovy goal, but until this occurs, check your Java code to make sure you can use it with the Groovy interpreter.

Statements

The first visible difference between Groovy code and Java code is that statements do not have to end with a semicolon. You have to use semicolons only to write several statements in one line. Of course, you can end every statement with a semicolon if you want. This feature does not stop you from spanning statements in multiple lines for better readability.

```
println “Statement without semicolon”
println “Statement with semicolon”;
print “Statement one! “; print “Statement two”
```

Some other programming languages make semicolons optional as well, and this has been a controversial topic for a long time. On one side are people who claim omitting semicolons saves you some typing and that this allows you to write code faster, but others say semicolons make code more readable and easier to debug. In the end, it boils down to what you’re used to, and because you are probably going to use Groovy together with Java (where semicolons are required), it is not likely that you are going to change your habits.

Loose Typing

As mentioned in Chapter 1, dynamic typing is one of the most important characteristics of scripting languages. To briefly summarize, dynamic typing does not force you to define the type for variables and properties, or for method arguments and return values. Types are automatically determined at the moment the value is assigned. They also could be changed later on, when the new value is assigned (or returned). We also introduced the term *weak typing*, which means variables are cast to the most suitable type before an operation takes place. Groovy implements both of these principles.

Take Listing 4.5 as an example.

Listing 4.5 Loose Typing

```
i = 10
println “i has type of “ + i.getClass()
s = “This is text”
println “s has type of “ + s.getClass()
```

Listing 4.5 Continued

```
i +=s
println i
println "now i has type of " + i.getClass()
```

The code in Listing 4.5 prints the following:

```
i has type of class java.lang.Integer
s has type of class java.lang.String
10This is text
now i has type of class java.lang.String
```

The variable `i` was automatically converted to the `String` type before the += operator was applied. If that conversion is not done, a `groovy.lang.MissingPropertyException` is thrown.

Dynamic typing shows its real strength when it is used with the *method overloading* and *polymorphism* principles. These are two of the most important principles of object-oriented programming.

Method overloading means we can have more methods of the same name in one class. These methods must have different parameter types so that the compiler can determine which method should be called. Briefly, polymorphism refers to the capability that one interface has many implementations.

In a strongly typed language, such as Java, you must define a common interface for objects that shares some of the properties and overload methods to handle different interfaces. In a scripting language such as Groovy, this is not necessary.

Look at Listing 4.6, for example.

Listing 4.6 Dynamic Typing and Polymorphism

```
class Car {
    public horsePower
    public color
}

class Shirt {
    public size
    public color
}

def printColor(item) {
```

Listing 4.6 Continued

```
    println "Item has " + item.color + " color"
}

car = new Car(color:"red")
shirt = new Shirt(color:"blue")
printColor(car)
printColor(shirt)
```

Here is the result of Listing 4.6's execution:

```
Item has red color
Item has blue color
```

Note that the `printColor()` function can work with any class that has the `color` property, and basically, we didn't have to define a common interface (or parent class) for the `Car` and `Shirt` classes to be able to handle both of them with one method. If the object does not have a `color` property, a `groovy.lang.MissingPropertyException` is thrown. Of course, this is not a call for avoiding the inheritance entirely, but just an example of how to do things that could not be done in Java.

Type Juggling

In Chapter 3, I explained the term *autoboxing* and how you can use it in Java and BeanShell. This feature is also supported in Groovy. Take, for example, the script in Listing 4.7.

Listing 4.7 Autoboxing

```
i = 12
System.out.println(i.getClass())

j = 12.0
System.out.println(j.getClass())

z = true
System.out.println(z.getClass())
```

This script prints the following:

```
class java.lang.Integer
class java.math.BigDecimal
class java.lang.Boolean
```

As we can see, Groovy treats everything as an object, which avoids tedious and hard-to-read casting between primitives and wrapper classes. Integral numerical literals are converted to the smallest type they fit in (`java.lang.Integer` in the preceding example).

The more interesting thing in the preceding example is that literals with decimal points are represented as the `java.math.BigDecimal` class and not as the `java.lang.Float` or `java.lang.Double` class. This is done to avoid problems with range, precision, and rounding of float and double values. With this so-called "least surprising" approach to literal math, you can avoid situations that could be unpleasant in Java. To demonstrate these problems, let's take a look at the following code:

```
public class Numbers {

    public static void main(String[] args) {
        System.out.println((2.11 + 0.11 == 2.22));
        System.out.println(2.11 + 0.11);
    }
}
```

This code prints the following:

```
false
2.2199999999999998
```

In Groovy, the equivalent script prints `true` because arbitrary precision arithmetic is used. If you want to explicitly use doubles or floats, you can always instance them directly, as follows:

```
j = new Double("12.0")
z = 12.0D

println z.class
```

The preceding example prints the following:

```
class java.lang.Double
```

You can see from the script that you can specify types using a suffix character. In this case, `D` is used to denote `java.lang.Double`. Table 4.1 provides a complete suffix definition.

Table 4.1 Suffix Definition

Suffix	Type
G	java.math.BigInteger
L	java.lang.Long
I	java.lang.Integer
G	java.math.BigDecimal
D	java.lang.Double
F	java.lang.Float

As you can see in Table 4.1, the `G` suffix is used for both the `BigInteger` and `BigDecimal` types, and which one is used depends on the actual value of the variable.

All the Java syntax for defining octals, hexadecimals, and numbers with exponents works in Groovy, too.

Arithmetic operations in Groovy tend to preserve floating-point numbers introduced in the operation. In other words, floats and doubles are not converted in `java.math.BigDecimal` during the operation.

Table 4.2 shows the result type of arithmetic operations for various operand types, excluding division operation.

Table 4.2 Arithmetic Operations Type Conversion

	BigDecimal	BigInteger	Double	Float	Long	Integer
BigDecimal	BigDecimal	BigDecimal	Double	Double	BigDecimal	BigDecimal
BigInteger	BigDecimal	BigInteger	Double	Double	BigInteger	BigInteger
Double	Double	Double	Double	Double	Double	Double
Float	Double	Double	Double	Double	Double	Double
Long	BigDecimal	BigInteger	Double	Double	Long	Long
Integer	BigDecimal	BigInteger	Double	Double	Long	Integer

Based on this table, we can expect that the following script:

```
i = 10.1F
j = 12.2D

z = i + j
println z.class
```

prints:

```
class java.lang.Double
```

Division has a special status, and it returns `java.lang.Double` if one of the operands is either `java.lang.Float` or `java.lang.Double`. In any other case, the result type is `java.math.BigDecimal`. If no scale is defined for operands, 10 is used as a default, and the result is normalized.

The `BigDecimal` arithmetic enables you to use fractions in Groovy naturally, as in the following example:

```
groovy> 1/2 + 1/3
0.8333333333
groovy> (1/2) / (1/4)
2
groovy> 1/2 * 2
1.0
```

Strings

As mentioned earlier, strings are one of the most important data types in scripting languages. So Groovy, as expected, treats strings in this manner and adds new capabilities to them.

To begin, you can define strings in Groovy using both single quote and double quote characters:

```
println “The ‘blue’ color is selected”
println ‘The “blue” color is selected’
```

The preceding code provides the following output:

```
The ‘blue’ color is selected
The “blue” color is selected
```

We can see that text quotations are now much easier to produce. If you are used to defining strings only with double quotes (or just with quotes), and you still want an easy way to escape those characters, you can use the following syntax:

```
println '''The 'blue' color is selected'''
println """The "blue" color is selected"""
```

This example produces the same output as the preceding example.

Strings in Groovy can also span multiple lines, so you can replace the following Java-like syntax:

```
String query =     "SELECT * FROM "
        +"table WHERE "
        +"field='value' ";
System.out.println(query);
```

with this more-elegant solution:

```
query = "SELECT * FROM \
    table WHERE \
    field = 'value'"
println query
```

The script prints the following:

```
SELECT * FROM
    table WHERE
    field = 'value'
```

We can see that the output retained the line breaks. Note that to enable Groovy to join multiple lines, you have to end each line with a backslash (\).

Another way to work with large strings and multiple lines is by using triple-quote syntax explained earlier. This is shown in Listing 4.8.

Listing 4.8 Triple-Quote Syntax

```
header = """
<html>
<head>
    <title>Title</title>
```

Listing 4.8 Continued

```
</head>
<body color="#FFFFF" font='Helvetica'>
    Hello!
</body>
</html>
"""

println header
```

Listing 4.8 shows us that with triple-quote syntax, we don't have to think about quotations because we used both single and double quotes. In Listing 4.8, the actual value of the *header* variable is marked in italics.

GStrings

Groovy also utilizes its own class, `groovy.lang.GString`, which enables you to embed expressions into a string. This enables `GString` objects to perform template processing and variable substitutions innately. The syntax for embedded expressions is `${expression}`, like the syntax used in the UNIX/Linux shell and in the Java Standard Tag Library (JSTL) expression language employed in Java Server Pages 2.0. All this further simplifies string manipulation in Groovy. GStrings are demonstrated in Listing 4.9.

Listing 4.9 GStrings

```
table = "users"
value = 10
query = "SELECT * FROM ${table} WHERE value = ${value}"
println query
```

When executed, the preceding script prints the following:

```
SELECT * FROM users WHERE value = 10
```

The expression could be any valid Groovy expression, not just the simple value. So the following code

```
table = "users"
value = null
query = "SELECT * FROM ${table} \
```

```
        ${(value == null) ? “” : “WHERE value =
${value}”}”
println query
```

would print the same output as the code from Listing 4.9, if the `value` variable was set to the value `10`. But it would print:

```
SELECT * FROM users
```

if the `value` variable hadn’t been defined (or was set to the `null` value).

The `GString` class uses the “lazy expression evaluation” method. In other words, expressions are not evaluated until the string value of the object is needed. At that moment, the `toString()` method is called an object, which evaluates expressions and returns the string value. So from a developer’s point of view, `GString` objects behave in the same way as regular strings. The only difference is their capability to embed source code directly into the value.

Regular Expressions

Regular expressions (regex) are a powerful tool for manipulating text and are used heavily in scripting languages. Regular expressions are used to match certain patterns in text. In addition, we can replace those patterns with new values. They have a wide range of uses, such as data validation and searching.

Since Java 1.4 Standard Edition (J2SE 1.4), regular expressions have been an integral part of the Java platform. I briefly go through regular expressions implementation in Java and then look at what Groovy has to offer on top of that.

Java regular expressions implementation is located in the `java.util.regex` package. Let’s start with an example of using regular expressions in Java:

```
Pattern p = Pattern.compile(“(ab)*”);
Matcher m = p.matcher(“ababababab”);
if (m.matches()) {
    System.out.println(“Data is valid”);
} else {
    System.out.println(“Data is not valid”);
}
```

The first thing to do in Java is compile a pattern and store it in the object of the `Pattern` class. We do not dig deeper into pattern syntax; you are advised to check with the appropriate literature if you are interested in that topic. The pattern we use in these examples simply matches the text that contains one or more `ab` literals in it. After that, we create the `Matcher` object containing the actual text we want to check against in the pattern. The `matches()` method of the `Matcher` object returns a boolean value indicating whether the text matched the given pattern. Because the text matches the pattern in this example, the earlier code snippet prints the following:

```
Data is valid
```

Of course, you can play with this example by changing the pattern and text, and see how the result depends on those changes.

Groovy treats regular expressions as an integral part of the language and introduces Perl-like syntax for handling them. Listing 4.10 is the Groovy alternative for the preceding example.

Listing 4.10 Regular Expressions

```
p = ~"(ab)*"
m = "abababab" =~ p
if (m.matches())
    println "Data is valid"
else
    println "Data is not valid"
```

As we can see, the following rules apply in Groovy:

- **~"pattern"**—Creates the `Pattern` object. It is used to replace `Pattern.compile("pattern")`.
- **"text" =~ pattern**—Creates the `Matcher` object. It is equivalent to `pattern.matcher("text")`.

You can also use more compact syntax for creating the `Matcher` object, such as the following:

```
m = "abababab" =~ "(ab"*"
```

where =~ is a replacement for `Pattern.compile("pattern").matcher("text")`. This is

convenient in cases where you don't need to use one `Pattern` on many `Matcher` objects.

When created, the `Matcher` object can be used in the standard Java manner. For example, we can use it to replace every subsequence of the input sequence that matches the pattern with the replacement string:

```
m = "abab" =~ "(ab)*"
println m.replaceAll("cd")
```

The preceding example replaces all the `ab` literals in the text with the `cd` literals and prints the modified text on the screen:

```
cdcd
```

Although the `Matcher` object is valuable for various tasks, regular expressions are commonly used just to validate some data. For example, we could check whether the user has entered an e-mail address in the correct format. For such validations, Groovy introduces the ==~ operator, which returns a Boolean value equivalent to the following Java code:

```
Pattern.compile("pattern").matcher("text").matches()
```

So, our beginning example could be rewritten as follows:

```
if ("abababab" ==~ "(ab)*")
    println "Data is valid"
else
    println "Data is not valid"
```

Collections

Collections represent objects that group multiple objects together into a single instance. Java collections are powerful, but they are treated as ordinary objects. This means that cumbersome code is required for their manipulation. Collections handling is one of the essential programming tasks, and therefore scripting languages tend to have language support for

different kinds of collections. Groovy follows this trend, so it has rich syntax to deal with all kinds of different methods for object grouping.

LISTS

Lists are ordered collections of elements associated with an integer index. When using lists, the programmer has precise control over the position of the element in the collection. Java uses the `java.util.List` interface for this collection type, which defines methods for basic list manipulation.

Groovy lists are also an implementation of the `java.util.List` interface. List creation is easy and similar to creating arrays in Java. Just write comma-separated values in the square brackets, as follows:

```
class Car {
    String model
}

list = ['groovy', 2, new Car(model:"VW")]
println list.get(2).model
println list[1]
```

As you can see, to access a particular element of the list, you could use Java syntax and call the `get()` method with the element's index as an argument. Another (more scripting oriented) way is to type the index of the element in the square brackets. When evaluated, this script prints the following output:

```
VW
2
```

The empty lists could be created using the `[]` expression. Just as with the `[]` operator used to access the list element with a certain index, Groovy adds the `<<` operator to simplify the addition of new elements in the list. Let's examine the following example:

```
groovy> list = []
[]
groovy> list.add(7)
[7]
groovy> list << 12
[7, 12]
```

The previous script creates an empty list and then adds two elements into it. One is added in the standard Java manner (the `add()` method) and the other using the << operator.

Groovy also extends the `java.util.List` interface with methods and operators, making list handling easier. Listing 4.11 demonstrates this feature.

Listing 4.11 Lists

```
groovy> list = [0,2,4,6]
[0,2,4,6]
groovy> list += [0,1,3,5,7]
[0,2,4,6,0,1,3,5,7]
groovy> list -= [0,1]
[2,4,6,3,5,7]
groovy> list.sort()
[2,3,4,5,6,7]
groovy> list.reverse()
[7,6,5,4,3,2]
groovy> list << 7
[7,6,5,4,3,2,7]
groovy> list.count(7)
2
groovy> "${list.min()} - ${list.max()}"
2 - 7
groovy> list.intersect([4,6,8])
[4,6]
groovy> list.join('-')
7-6-5-4-3-2-7
```

As we can see in Listing 4.11, the following methods and operators have been added to the `java.util.List` interface:

- **+ (plus sign)**—Creates the union of two lists. Note that duplicated elements are not removed after this operation.
- **- (minus sign)**—Removes all elements specified in the list on the right side of the operator.
- **`sort`**—Sorts the list. This method could accept the `java.util.Comparator` or `groovy.lang.Closure` object as a parameter. We return to this topic after introducing closures in Groovy, later in this chapter.
- **`reverse`**—Reverses the list ordering.
- **`count`**—Returns the number of occurrences for a given element.
- **`min/max`**—Returns the minimal (or the maximal) element of the list. As with the `sort()` method, this method

could accept the arbitrary `Comparator` or `Closure` object.

- **`intersect`**—Returns a list of common elements for two lists.
- **`join`**—Returns the string value of the list elements concatenated with the given string.

RANGES

Another new feature in Groovy collections processing is the use of ranges, which can be very useful in everyday programming tasks. Ranges are basically lists of sequential values. You can declare both inclusive and exclusive ranges with the following syntax:

```
groovy> inclusive = 7..19
7..19
groovy> exclusive = 7..<19
7..18
```

As you can see, the `..` expression is used to create the range with the last boundary value included. On the other hand, if you use the `..<` expression, the last boundary value is not included in the range.

Ranges are implemented as a `java.util.ArrayList` class, but if `Integer` objects are used for boundaries, the `groovy.lang.IntRange` class is used. This class is the lightweight implementation of the `java.util.List` interface that holds only the boundary values. All other elements are calculated at runtime according to their indexes. You can define range with objects of any type, but they make sense only when these objects implement the `java.lang.Comparable` interface.

A few methods also are added to the ranges that make them easier and more natural to handle. They are demonstrated in Listing 4.12.

Listing 4.12 Ranges

```
groovy> inclusive = 5..10
5..10
groovy> inclusive.getFrom()
5
```

Listing 4.12 Continued

```
groovy> inclusive.getTo()
10
groovy> inclusive.contains(7)
true
```

These methods are as follows:

- **getFrom()**—Gets the starting boundary of the range
- **getTo()**—Gets the ending boundary of the range
- **contains()**—Checks whether the given element is in the range

Ranges play an important role in Groovy programming philosophy. We see their real power in the "Looping" section later in this chapter. For now, let's take a look at an interesting use of ranges for slicing lists and strings:

```
groovy> list = ["groovy", "python", "beanshell", "ruby"]
["groovy", "python", "beanshell", "ruby"]
groovy> sublist = list[1..2]
["python", "beanshell"]
groovy> lang = sublist[1]
"beanshell"
groovy> lang[4..8]
"shell"
```

In this example, we began by creating a list of four string elements. After that, we created a sublist containing the two middle elements using the range syntax. Finally, we sliced one element (of the string type) with ranges too, and showed that strings could be seen as a list of characters.

MAPS

Maps, though not strictly considered collections in the mathematical sense, are often associated with collections processing in languages such as Java. In lists, elements of the collection are associated with a numeric position; in maps, elements are accessed by names, also known as keys. Maps in Groovy are instances of the `java.util.Hashmap` class. As with lists, maps in Groovy have powerful syntax enhancements that make them easier to use.

```
groovy> book = ["name" : "Scripting in Java"
              , "author" : "Dejan Bosanac"]
["name":"Scripting in Java", "author":"Dejan Bosanac"]
groovy> book.get("name")
"Scripting in Java"
groovy> book["author"]
"Dejan Bosanac"
groovy> book.topic = "Java"
"Java"
groovy> book
["name":"Scripting in Java", "topic":"Java"
, "author":"Dejan Bosanac"]
```

As we can see in the preceding code, maps also are created using square-bracket syntax. The only difference between maps and lists is that with maps, elements are provided with their keys and values separated by colons. Also, as the preceding example shows, a map element can be accessed in one of three ways:

- Using the Java syntax (the `get()` and `put()` methods)
- Using the list syntax (the `[]` operator)
- Using the JavaBean syntax (the . operator)

An empty map is created using the `[:]` expression, as shown in Listing 4.13.

Listing 4.13 Maps

```
groovy> book = [:]
[:]
groovy> book.name = "Scripting in Java"
"Scripting in Java"
groovy> book["topic"] = "Java"
"Java"
groovy> book.put("author", "Dejan Bosanac")
null
groovy> book
["name":"Scripting in Java", "topic":"Java"
, "author":"Dejan Bosanac"]
```

The preceding example uses a string variable as the key type for the map. Of course, any object could be used as a key, but to comply with the JavaBean syntax, keys should always be strings. If we tried to use the JavaBean syntax on the map with nonstring keys, an exception is thrown:

```
groovy> testMap = [ 1 : "first", 55 : "second"]
[1:"first", 55:"second"]
groovy> testMap.get(1)
"first"
groovy> testMap[55]
"second"
groovy> testMap.55
No signature of method java.util.HashMap.doCall() is
applicable
 for argument types: (java.math.BigDecimal) values: [0.55]
```

Logical Branching

As shown in the following example, Groovy supports an `if-else` structure and a ternary operator in the same way Java does, so we do not describe them in more detail.

```
x = true

if (x)
    println "x is true"
else
    println "x is false"

println "x is " + (x == true ? "true" : "false")
```

The new thing that Groovy has to offer developers is a modified and more powerful `switch-case` structure. Let's take a look at an example (see Listing 4.14).

Listing 4.14 `switch-case` **Structure**

```
switch (x) {

 case ~"\\d{5}"           : println "it's a zip code"
                              break
 case ["Groovy", "Java"] : println "it's a programming language"
                              break
 case String              : println "it's a java.lang.String"
                              break
 case 150                 : println "it's a 150$"
                              break
 case 100..200            : println "it's between 100$ and 200$"
                              break
 default                  : println "unknown"

}
```

As we can see in Listing 4.14, Groovy can compare a `switch` variable with the following criteria:

- Class name
- Regular expressions
- Membership in the collection
- Exact value

You can try to “feed” this script with different values for the variable `x`, and see what happens:

```
x = “any string”
x = 90210
x = “Groovy”
x = 150
x = 180
```

The real improvement in Groovy is that the `switch-case` structure behavior is extensible. It is important to understand the structure’s underlying mechanism to be able to customize it to your needs. When code such as the following is being executed:

```
switch (switchValue) {
    caseValue : //action
}
```

Groovy calls the `isCase(Object obj)` method on the `caseValue` object with `switchValue` as an argument, like this:

```
caseValue.isCase(switchValue)
```

The `isCase()` method is overloaded for many classes by default. For example, for the `java.lang.Class` type, it looks like this:

```
public boolean isCase(switchValue) {
    return switchValue instanceof this.class
}
```

If no `isCase()` method is found, a default one is called. The default `isCase()` method implementation simply calls the `equals(Object obj)` method on the switch value.

In this light, let's take a look at Listing 4.15.

Listing 4.15 Extending the `switch-case` Mechanism

```
class Dog {
    public isCase(switchValue) {
        if ([“labrador”, “shepherd”]
            .contains(switchValue))
            return true
        else
            return false
    }
}

def testAnimal(animal) {
    doggy = new Dog()
    switch (animal) {
        case doggy :    println “${animal} is a dog”
                         break
        default    :    println “${animal} is not a dog”
    }
}

testAnimal(“labrador”)
testAnimal(“piggy”)
```

The script in Listing 4.15 prints the following sentences:

```
labrador is a dog
piggy is not a dog
```

We defined a `Dog` class with its own `isCase()` method that returns `true` if the switch value is a string contained in the list of dog breeds. In the `testAnimal()` method, the string has been tested with this overloaded `isCase()` method.

As we have seen, the `switch-case` structure extension could be useful for handling specific data types.

Looping

As is the case with logical branching, the `while` and `do-while` loops are backward compatible with Java, so a detailed explanation is not required. Here is a simple script that demonstrates these loops:

```
x = 0
y = 10
```

```
while (x++ < 10) {
    println y--
}

x = 0
y = 10

do {
    println y--
} while (++x < 10)
```

Things are somewhat different with the `for` loop. As mentioned previously, Groovy still lacks support for the standard Java `for` loop, which is one of the things that break backward compatibility. Instead, it has a flavored `for` loop that more naturally fits different types. The use of this loop for various tasks is demonstrated in Listing 4.16.

Listing 4.16 `for` Loop

```
println "iterate over a range"
x = 0
for ( i in 0..9 ) {
    print i
}

println "\n iterate over a list"
x = 0
for ( i in [0, 1, 2, 3, 4] ) {
    print i
}

println "\n iterate over an array"
array = (0..4).toArray()
x = 0
for ( i in array ) {
   print i
}

println "\n iterate over a map"
map = ['abc':1, 'def':2, 'xyz':3]
x = 0
for ( e in map ) {
    print e.value
}

println "\n iterate over values in a map"
x = 0
for ( v in map.values() ) {
    print v
}

println "\n iterate over the characters in a string"
text = "abc"
```

Listing 4.16 Continued

```
list = []
for (c in text) {
    print c + " "
}
```

Listing 4.16 prints the following output:

```
iterate over a range
0123456789
iterate over a list
01234
iterate over an array
01234
iterate over a map
132
iterate over values in a map
132
iterate over the characters in a string
a b c
```

As you can see, the syntax of this `for` loop is:

```
for (var in structure) {
    do something ...
}
```

where `var` is an element of the structure, which can be merely any structure in Groovy (including strings).

The syntax of this loop is similar to that for the `for` loop introduced in Java 1.5:

```
List<String> values = new ArrayList<String>();
names.add("a");
names.add("b");
names.add("c");

for (String value: values)
   System.out.println(value);
```

The only difference is that Java's `:` character is replaced with the `in` keyword in Groovy. Also, because Java is statically typed, the element definition in the loop contains the element's type.

Classes

Classes in Groovy are similar to Java classes; after all, they are compiled to Java classes at the bytecode level. Still, there are a few differences that save on typing and add more flexibility.

Following its loose typing philosophy, Groovy does not force you to define types for properties, method arguments, and return values. If the type is not specified, the `java.lang.Object` class is used at the bytecode level.

Also, you don't need to define access modifiers for class members. If no access modifier is specified, Groovy assumes `protected` access level.

Because both type definition and access modifier are optional, you have to use the `def` keyword for class members lacking both of these properties.

Now take a look at Listing 4.17.

Listing 4.17 Classes

```
Class Book {
    def title
    String author
    public Book(title) {
        this.title = title;
    }

    boolean order(int qty, Reseller) {
        true
    }

    def title() {
        "Book title"
    }
}
```

The Groovy code in Listing 4.17 is compiled to the following Java code:

```
class Book {
    public Object title;
    public String author;

    public Book(Object title) {
        this.title = title;
    }

    public Object order(int qty, Object reseller) {
```

```
        return true;
    }
    public Object title() {
        return "Book title";
    }
}
```

Notice that the `return` statement at the end of the method is also optional. The value of the last statement would be automatically returned if that line were reached. Of course, if you want to return from some other point in your method, you use the `return` keyword.

Parentheses around parameters in method calls are optional for all methods except constructor calls:

```
book = new Book("Scripting in Java")
// mandatory parentheses in the constructor calls
book.order 5, "BookReseller inc." // without parentheses
book.order(5, "Books Ltd.") // with parentheses
```

This is true as long as the method has at least one parameter because the method call could be mistaken for the class's property otherwise. This issue is demonstrated in the following code snippet:

```
book.title // gets property value
book.title() // calls the method
```

Standalone methods, defined with the `def` keyword, follow the same rules as class methods for issues described earlier. As mentioned previously, they are basically static methods of the class named as our script.

The same discussion about the optional `return` keyword and parentheses stands as for the optional semicolons mentioned previously. Altering coding habits probably requires more work for Java developers than what it's worth in terms of benefits gained from added features. However, it is important to know them so that you can deal with someone else's Groovy code.

Groovy shows its dynamic nature once again in the way it handles calls to an unimplemented class method. All objects defined in Groovy implement a `groovy.lang.GroovyObject`

interface, which helps them to be executed in the Java environment. This interface has defined the following method:

```
Object invokeMethod(String name, Object args)
```

This method is invoked every time a call to some method is made. You can override this method to extend your class's functionality and intercept calls to its methods, as shown in Listing 4.18.

Listing 4.18 Intercepting Method Calls

```
class MyClass {

     public String callMe(String arg) {
          return arg.toUpperCase()
     }

     public invokeMethod(String name, Object arguments) {

          if (name == "me") name = "callMe"

          try {
               return metaClass.invokeMethod(this, name, arguments)
          } catch (MissingMethodException e) {
               return "Method ${name}:${arguments} is not defined \
                         for this class"
          }
     }
}

o = new MyClass()

println o.callMe("Some String")
println o.callMe("Some String", "Some Other String")
println o.me("Some String")
```

The code in Listing 4.18 intercepted a call to a `me()` method and called the `callMe()` method instead. Also, the call to the `callMe()` method with two parameters returns a string result instead of throwing a `MissingMethodException`. The result of this script's execution is as follows:

```
SOME STRING
Method callMe:[Some String, Some Other String] is not
defined
for this class
SOME STRING
```

Operator Overloading

To support scripting syntax better, Groovy provides the operator overloading mechanism. We have already seen some examples of the operator overloading mechanism in the "Collections" section earlier in this chapter. It means that you can use some of the standard operators, such as + or -, with your classes. These operators are mapped to method calls of objects. So, for example, the following:

```
a+b
```

is mapped to the actual call:

```
a.plus(b)
```

The purpose of operator overloading is to make your scripts more readable and to save you some typing.

There are a few operator types you can use:

- **Arithmetic operators**—Used to execute arithmetic operations on objects
- **Index operators**—Used to access elements in collections of objects
- **Shift operators**—Used to manipulate data by shifting
- **Comparison operators**—Used to compare objects

Table 4.3 describes all operators currently supported by Groovy syntax.

Table 4.3 Operator Overloading

Operator	Method
Arithmetic operators	
`a + b`	`a.plus(b)`
`a - b`	`a.minus(b)`
`a * b`	`a.multiply(b)`
`a / b`	`a.divide(b)`

Table 4.3 Continued

Operator	Method
Indexing operators	
a++ or ++a	a.next()
a– or –a	a.previous()
a[b]	a.getAt(b)
a[b] = c	a.putAt(b, c)
Shift operators	
a << b	a.leftShift(b)
Comparison operators	
a == b	a.equals(b)
a != b	! a.equals(b)
a === b	a == b in Java (a and b refer to same object instance)
a <=> b	a.compareTo(b)
a > b	a.compareTo(b) > 0
a >= b	a.compareTo(b) >= 0
a < b	a.compareTo(b) < 0
a <= b	a.compareTo(b) <= 0

These operators are already implemented in many Groovy data structures, such as collections and numbers. For example, the last two calls in Listing 4.19 are basically the same.

Listing 4.19 Operators

```
groovy> list = [0, 3, 5, 7]
groovy> list.getAt(2)
5
groovy> list[2]
5
```

Another important thing is comparison operators handle `null` values gracefully. In other words, if one (or both) values are `null`, no `java.lang.NullPointerException` is thrown:

```
groovy> first = null
groovy> second = “test value”
groovy> if (first == second)
groovy>     println “they are equal”
groovy> else
```

```
groovy>     println "they are not equal"
groovy> go
they are not equal
```

When you are working with numbers, values are converted to the largest numeric type before the comparison is made. This feature enables developers to compare different number types; the only thing that matters in this comparison is their values:

```
groovy> Double first = 123.50
groovy> Integer second = 250
groovy> if (first > second)
groovy> println first
groovy> else
groovy> println second
groovy> go
250
```

The benefits of the operator overloading mechanism can be used with custom classes. Look at the code in Listing 4.20.

Listing 4.20 Operator Overloading

```
class Item {
}

class Basket {
    Item items

    public plus (Item item) {
        println "Adding item"
        // adds item to the basket and recalculates it
    }
}

basket = new Basket()
item = new Item()
basket + item
```

The `plus()` method is added to the `Basket` class, and after that, the + operator is available. This can be done with any operator on any class. Note that the `equals(Object obj)` method is defined in the `java.lang.Object` class, so operators that map to it can be used on any object by default. Also, the `compareTo(Object obj)` method is defined in the `java.lang.Comparable` interface, so it is advisable to implement this interface for objects that have to work with comparison operators.

GroovyBeans

Groovy introduces a few changes in JavaBeans syntax. All changes are aimed to help developers code their solutions faster.

PROPERTIES

The first major difference is properties and methods are public by default and are not protected (see Listing 4.21).

Listing 4.21 GroovyBeans

```
class Car {
    String model
    String color
    private Integer id
    Integer year

    private setYear(Integer year) {
        this.year = year
    }
}
```

At the virtual machine level, things are more complicated. Groovy compiles public properties to private properties with public accessor methods (*getter/setter*). Protected properties are also compiled to private properties but with protected accessor methods. For private properties, no *getter/setter* methods are created on the bytecode level.

If we follow this concept, our `Car` class definition in the preceding code is equivalent to the following JavaBeans definition:

```
class Car {
    private String model;
    private String color;
    private Integer id;
    private Integer year;

    public String getModel() {
        return this.model;
    }

    public String getColor() {
        return this.color;
    }

    public Integer getYear() {
        return this.year;
    }
```

```
    public void setModel(String model) {
        this.model = model;
    }

    public void setColor(String color) {
        this.color = color;
    }

    private void setYear(Integer year) {
        this.year = year
    }
}
```

Obviously, Groovy's solution is more elegant and faster to implement. If you need more control of the class's property behavior (for example, to create a read-only property), you can achieve it by explicitly declaring the suitable accessor methods. Look at the read-only property `year` in our `Car` example. We specified that the `setYear()` method is private and thus overloaded the auto-generated public version of this method. In this way, we created the read-only property.

Named Parameters

Another type-saving feature in Groovy is named parameters. GroovyBeans can be constructed by passing property names and values separated by a colon character in the constructor call. For example, we can create an instance of the `Car` class defined earlier with the following statement:

```
car = new Car(model:"BMW", color:"black")
```

This statement is compiled to the code that has an empty constructor call and appropriate *setter* method calls. Thus, it is equivalent to the following Java code:

```
Car car = new Car();
car.setModel("BMW");
car.setColor("black");
```

The named parameters mechanism could also be used for methods that accept map objects as a parameter:

```
def mtd(dog) {
    println dog.name
    println dog.breed
```

```
}

mtd(name : “Lina”, breed : “Labrador”)
mtd([“name” : “Lina”, “breed” : “Labrador”])
//the same as above
```

This approach is applicable only to maps that have string keys, and as you can see, it is just a shorter way to define that kind of map.

OBJECT NAVIGATION

You can access GroovyBeans’ properties in the same way as if they were public fields, so the following statement

```
car.name
```

is equivalent to this JavaBeans call:

```
car.getName()
```

More complex object navigation works in the same way; just use the dot (.) character to specify a desired property:

```
class Car {
    Manufacturer man
    String model
}

class Manufacturer {
    String name
    String country
}

man = new Manufacturer(name:”BMW”, country:”Germany”);
car =  new Car(man:man, model:”316i”)

println “Car origin is in ${car.man.country}”
```

This method of object navigation works fine unless some property in the chain is not defined. If such a condition is met, as in the following example:

```
car = new Car(model:”320i”)
println “Car origin is in ${car.man.country}”
```

a `java.lang.NullPointerException` is thrown.

SAFE NAVIGATION

To avoid `NullPointerExceptions` while navigating GroovyBeans, you can use Groovy's safe navigation syntax.

To safely navigate through beans, just use the `?.` operator rather than the `.` operator. In normal circumstances, you do not see any difference, but if some property in the chain is not defined, the `null` value is returned (instead of throwing the `NullPointerException`). Look at Listing 4.22, for example.

Listing 4.22 Safe Navigation

```
car = new Car(model:"320i")
println "Car origin is in ${car?.man?.country}"
```

In this case, the `null` value is returned, and there is no fear of `java.lang.NullPointerException`.

Whether to traverse your objects using the standard GroovyBean syntax or the safe navigation syntax is a design decision based on the following question: Should we handle the null pointer error at this point (in which case, we would use the GroovyBean syntax inside the appropriate `try/catch` block), or should we just ignore it (in which case, we should use the safe navigation syntax)?

Closures

We discussed closures in Chapter 1, where we also saw the benefits they introduce in the development process. Groovy supports closure syntax similar to that found in the Ruby programming language, and that is certainly one of the most important Groovy features. Let's look at a few examples of using closure in Groovy.

```
logo = {
    print "Closures "
    println "rule!"
}

logo.call()
logo()
```

The preceding example shows how to define and call a simple closure. We created a `logo` closure, and all we had to do was write statements inside the curly braces. The closure can be called by executing a `call()` method on it, or by calling it as a regular standalone method.

You can also pass parameters to closures. If only one parameter is passed, the closure's definition remains the same, and that parameter is mapped to the `it` variable. This is demonstrated in the following example:

```
groovy> discount = { it * 0.8}
groovy> discount(200)
160.0
```

The syntax for closures that accept more than one parameter is different:

```
{ comma-separated-parameter-list -> statements }
```

Let's discuss this on a real example:

```
groovy> totalPrice = {subtotal, tax, discount ->
groovy>    taxDiscount = subtotal * discount
groovy>    taxDiscount * (1 + tax)
groovy> }
groovy> totalPrice(100, 0.2, 0.3)
36.0
```

Here, we defined a closure that accepts three parameters (`subtotal`, `tax`, and `discount`) and called it afterward. The only difference is that now we have a comma-separated list of parameters at the beginning of the closure (separated from the rest of the statements by the -> character).

If you pass the wrong number of parameters to the closure, an `IncorrectClosureArgumentException` is thrown. This is not the case with the `it` variable, which is assumed to have a `null` value in that case.

Thus far, we have seen how to define and execute closures, and you are probably wondering how it is different from loosely defined methods. A big difference is that closures are basically classes that extend the `groovy.lang.Closure` class.

Closures' statements are collected in a `call()` method, which is (as we have seen) one way to execute them. Because they are classes, they can be passed around as arguments to other methods. So basically, we are now able to pass code snippets around our application. Let's further elaborate on this using the code in Listing 4.23.

Listing 4.23 Closures

```
class Handler {
     def action
     def handle(object) {
          action(object)
     }
}

log = {object -> println "Action occurred: ${object}"}
save = {object ->
     // ... save object to the database
     println "Object saved to the database: ${object}"
}

logHandler = new Handler(action:log)
saveHandler = new Handler(action:save)

obj = "Status changed"
logHandler.handle(obj)
saveHandler.handle(obj)
```

This script prints the following result:

```
Action occurred: Status changed
Object saved to the database: Status changed
```

In the `Handler` class, an `action` property is defined that contains a closure with the explicit code to be executed. The code could be different on an object basis and not only on a class basis, which is the case with inheritance. After that, we defined two closures, one that only logs the object (in this simple example on the screen) and one that saves the object in the database. `Handler` objects are instanced with an `action` parameter, and we actually changed their behavior without making any subclasses. As in the discussion on loose methods, this is more suitable for simple problem solutions, and you should still hang on the inheritance where you find it appropriate.

One thing that differentiates closures from anonymous inner classes in Java is the variables' scope. In Groovy, variables defined in the context where a closure is created could be used and modified inside that closure. Also, any variable defined within a closure is visible in the surrounding context. Take a look at the following code snippet for a demonstration of this concept:

```
tax = 0.2
cl = {
    if (tax == 0.2) {
        tax += 0.1
        discount = 0.2
    } else {
        discount = 0.3
    }
}

cl()
println tax
println discount
```

A global `tax` variable is used in the closure, and its value has been changed. Also, we accessed the `discount` variable defined in the closure. The result of this script execution is as follows:

```
0.3
0.2
```

Closures show their real power when they are integrated with collections. Lists and maps in Groovy have additional methods that accept closure arguments. Their purpose is to be used as a replacement for Java's way of iterating collections. As mentioned earlier, strings in Groovy can be used as lists of characters, so the methods described in the following subsections apply to them as well.

each

This method iterates through the collection supplying the executable code in the form of a closure, which is executed on the `each` element of the collection. This is the literal alternative to the `java.util.Iterator` class. Listing 4.24 shows an example.

Listing 4.24 The each() Method

```
groovy> sum = 0
groovy> list = [3,5,7,9]
groovy> list.each{ sum += it }
groovy> sum
24

groovy> bookDef = “”
groovy> book = [“name” : “Scripting in Java”, “topic” : “Java”]
groovy> book.each{ bookDef += “${it.key}: ${it.value}\n” }
groovy> bookDef
name: Scripting in Java
topic: Java

groovy> value = “Groovy rulez!”
groovy> newValue = “”
groovy> value.each{ newValue += it.toUpperCase() }
groovy> newValue
GROOVY RULEZ!

groovy> fact=1
groovy> range = 1..7
groovy> range.each { fact *= it }
groovy> fact
5040
```

Listing 4.24 demonstrates how to use the each() method on lists, maps, strings, and ranges.

Curly Braces

When you are passing a closure to a method, the starting curly brace ({) must be in the same line as the method call. So, for example, the following snippet is not valid Groovy code:

```
[1,2,3].each
{
    println it
}
```

However, if you want to specify the starting curly brace on a separate line, you can use the following syntax:

```
[1,2,3].each(
{
    println it
}
)
```

collect

The `collect()` method transforms elements of a collection using the given closure (see Listing 4.25).

Listing 4.25 The `collect()` Method

```
groovy> list = [1,2,3,4,5]
groovy> doubles = list.collect{ it * 2}
[2,4,6,8,10]
```

You cannot achieve the same behavior with the `each()` method because it does not change the original collection. The `collect()` method returns a new collection containing the modified items.

inject

If you need to pass the result from the previous iteration to the next one, use the `inject()` method, as shown in Listing 4.26.

Listing 4.26 The `inject()` Method

```
list = [3,5,7,9]
list.inject(0) { prevItem, item ->
    println "${prevItem} - ${item}"
    return item
}
```

This script prints the following result:

```
0 - 3
3 - 5
5 - 7
7 - 9
```

As you can see, the syntax of the `inject()` method is different from the syntax of the previously described methods:

```
Collection.inject(firstInjectValue) {
injectedValue, collectionItem ->
        // closure code
}
```

To better understand this injection syntax, let's rewrite the preceding example as follows:

```
list = [3,5,7,9]
closure = {prevItem, item ->
    println "${prevItem} - ${item}"
    item
}
list.inject(0, closure)
```

We see that that the `inject()` method basically accepts two parameters: the injection value used in the first iteration and the closure to be used. The closure must define two parameters as well: The first one is for the value injected from the previous iteration, and the second one is for the collection item. Note that the closure has to return a value injected in the next iteration. Otherwise, the `null` value is assumed.

find

To find the first occurrence of a collection's item that meets a certain criterion, we can use the `find()` method. Take Listing 4.27, for example.

Listing 4.27 The `find()` Method

```
list = [3,5,7,9]
found = list.find{ it > 5 }
println found
```

The code in Listing 4.27 prints the value 7. Note that the closure must return `true` to make a match. If no item matches the given criteria, the `null` value is returned.

findAll

Unlike the `find()` method that returns only the first item, the `findAll()` method returns a collection of all items meeting the criteria defined by a closure. In Listing 4.28, I changed the example shown in Listing 4.27 to use the `findAll()` method.

Listing 4.28 The `findAll()` Method

```
list = [3,5,7,9]
found = list.findAll{ it > 5 }
println found
```

The code in Listing 4.28 prints the following:

```
[7,9]
```

The same restrictions on the closure's return value stand as for the `find()` method.

every

This method checks whether every item of the collection meets the criteria given by a closure. The method returns a Boolean value, which indicates whether we have a match on all the items. Listing 4.29 shows an example.

Listing 4.29 The every() Method

```
groovy> list = [3,5,7,9]
groovy> found = list.every{ it > 5 }
false
```

any

As its name implies, the `any()` method returns a boolean value indicating whether any of the items meet the criteria (see Listing 4.30).

Listing 4.30 The any() Method

```
groovy> list = [3,5,7,9]
groovy> found = list.any{ it > 5 }
true
```

As I said for the `find()` and `findAll()` methods, closures in the `every()` and `any()` methods should return a Boolean value for these methods to make any sense. If the value is not of a Boolean type, type conversion will be used, and the Boolean value will be evaluated.

Another thing that closures provide is an improved way of dealing with complex `try/catch/finally` structures. With closures, it is easy to write code that properly handles resources and exceptions. For example, new methods have been added to standard Java classes that handle files, processes, and database connections. When they are used in Groovy, you don't have to worry about closing resources. We examine these methods next, but before we do, it is a good idea to see the principles of their implementation. These principles can be used to adapt almost

any Java library that works with resources to Groovy's programming philosophy. Let's say that we have a class in a library that has something to do with resources:

```
class Resource {

    public Resource(String resourceName)
        throws ResourceException {
        // open the resource
    }

    public Object read() throws ResourceException {
        // return data or false as the end marker
    }

    public void close() throws ResourceException {
        // close the resource
    }
}
```

Typical Java code that opens, reads, and closes the resource looks like this:

```
Resource res = new Resource("someName");

try {
    while (result = res.read()) {
        println(result);
    }
} catch (ResourceException e) {
    e.printStackTrace();
} finally {
    try {
        res.close();
    } catch (ResourceException e) {
    }
}
```

This is a somewhat clumsy solution, especially when your application handles a lot of resources. A common solution is to make a wrapper class that does the work for you. Listing 4.31 shows the new `read()` method that has been added to the original class and enables easy resource processing with closures.

Listing 4.31 Resource Handling with Closures

```
class Resource {

    public Resource(String resourceName)
        throws ResourceException {
        // open the resource
```

Listing 4.31 Continued

```
    }

    public Object read() throws ResourceException {
        // return data or false as the end marker
    }

    public void read(String resourceName, Closure closure)
        throws ResourceException {
        open(resourceName);
        try {
             while (result = res.read()) {
                 closure.call(result);
             }
        } catch (ResourceException e) {
            throw e;
        } finally {
            try {
                close();
            } catch (ResourceException e) {
            }
        }
    }

    public void close() throws ResourceException {
        // close the resource
    }
}
```

Now all that the programmer needs to do to achieve the same functionality as the preceding Java solution is the following:

```
new Resource("someName").read {
    println it;
}
```

Exception handling, resource opening, and closing have been moved to the `read()` method that accepts a closure parameter. This is a common philosophy that has been used often in Groovy. It makes Groovy scripts more readable, but also makes them quicker and easier to write.

The preceding method is applicable when you have a library source, but if you want to extend a closed library or Java's built-in classes, you must use a different approach, which we cover later in this chapter.

System Operations

Closure support has been added to standard Java classes that operate on files and processes. These modifications are visible from Groovy as new methods (functionalities) added to base Java classes. In this section, we go through these methods and see some examples of how you can use files and processes in Groovy, and how closures can help simplify some of these tasks.

Files

The `java.io.File` class has been extended with a few methods that accept closure parameters and simplify file operations. These methods are implemented according to the principles for handling resources and exceptions described in the previous section. Let's go through some examples.

getText

This method returns the entire contents of the file as a string. It is useful for handling text files.

```
import java.io.File

text = new File("foo.txt").getText()
print text
```

As you can see, no `Reader` object instances, no closing statements, and no exception handling are required. All we have to do is write just the business logic of the script, and Groovy takes care of the rest.

Of course, if the file is not found, or some other exception occurs, the script throws a `java.io.FileNotFoundException`. You can handle exceptions on your own, but you still don't have to worry about resources. Take Listing 4.32, for example.

Listing 4.32 The `getText()` Method

```
import java.io.File

try {
    text = new File("foo1.txt").getText()
    print text
```

Listing 4.32 Continued

```
} catch (Exception e) {
    println e.getMessage()
}
```

Listing 4.32 prints the following if `foo1.txt` file could not be found in the current directory:

```
java.io.FileNotFoundException:
    foo1.txt (No such file or directory)
```

eachLine

Listing 4.33 shows how to open and read every line of the file. Particularly, the closure passed to this method prints every line of the file in uppercase.

Listing 4.33 The `eachLine()` Method

```
import java.io.File

new File("foo.txt").eachLine { println it.toUpperCase() }
```

readLines

This method, shown in Listing 4.34, has the same purpose as the `eachLine()` method, but instead of taking closure as an argument, it returns lines of the file in the list.

Listing 4.34 The `readLines()` Method

```
import java.io.File

lineList = new File('foo.txt').readLines()
lineList.each {
    println it.toUpperCase()
}
```

We can use some of the methods for list handling to operate on loaded data. In this example, we used the `each()` method and created the equivalent functionality as in the preceding `eachLine()` example.

splitEachLine

Groovy tends to make common tasks as simple as possible. In that context, the `java.io.File` class has an additional method used for working with comma-separated files (the other characters can be used as separators as well). This method enables you to handle CSV files with just a few lines of code. Take a look at Listing 4.35.

Listing 4.35 The `splitEachLine()` Method

```
import java.io.File

new File('foo.csv').splitEachLine(',') {
    it.each{
        println "name=${it[0]} balance=${it[1]}"
        // ... save into database
    }
}
```

If we feed this script with the following `foo.csv` file:

```
dejan,200
joe,100
mike,500
```

it prints text similar to the following:

```
name=dejan balance=200
name=joe balance=100
name=mike balance=500
```

Here we parsed the comma-separated file with just two lines of code. We finish this example in Chapter 5, after we cover the GroovySQL extension. We use data parsed from this file and insert that into the database.

eachByte

When you are dealing with binary files, you need access to the file on the byte level. This method is equivalent to the `eachLine()` method, except that it passes each byte to the closure for manipulation (see Listing 4.36).

Listing 4.36 The eachByte() Method

```
import java.io.File

new File("foo.txt").eachByte { print it }
```

readBytes

As with the eachByte()method, readBytes() is equivalent to readLines()on the byte level, as shown in Listing 4.37.

Listing 4.37 The readBytes() Method

```
lineList = new File('foo.txt').readBytes()
lineList.each {
    print it
}
```

write

Listing 4.38 shows how to write text to the file.

Listing 4.38 The write() Method

```
new File('foo.txt').write("testing testing");

new File('foo.txt').write("""
This is
just a test file
to play with
""");
```

In this example, we used triple-quote syntax explained in one of the previous sections. You can see how triple-quote syntax can ease the task of writing multiple-line strings in the file. Instead of formatting the text manually, all you have to do is define multiple-line strings inside of the appropriate markers.

Be careful with this method because it overwrites the contents of the file.

append

The append() method adds specified text to the end of the file. Other than that, it behaves exactly the same as the previously described write() method (see Listing 4.39).

Listing 4.39 The append() Method

```
new File('foo.txt').append("""
testing testing
""");
```

eachFile

Because the `java.io.File` class is used to represent directories as well, this method is used for processing all files in the directory. Each file in the directory is passed to the closure argument. Listing 4.40 prints them all on the screen.

Listing 4.40 The eachFile() Method

```
import java.io.File

new File('.').eachFile {
    println it.getText()
}
```

eachFileRecurse

This method is different from the previous one, only because it traverses directories recursively. Listing 4.41 shows an example of this method.

Listing 4.41 The eachFileRecurse() Method

```
import java.io.File

new File('.').eachFileRecurse {
    println it.getText()
}
```

If you want to handle files using reader and writer objects, as in Java, you can find some helper methods in Groovy that help you to obtain them. For more information on these methods, see the Groovy JDK reference (http://groovy.codehaus.org/groovy-jdk.html).

Processes

Groovy also provides closer integration with the native system processes. This integration is made by an additional `execute()` method of the string class:

```
process = "ls -l".execute()
println process.text
```

The method returns the `java.lang.Process` class (just like Java's `Runtime.exec()` method). The `Process` class has also been extended with the `getText()`method, used to obtain the text from the output stream:

```
process = "ls -e".execute()
error = process.err
if (process.waitFor() != 0)
    error.eachLine {
        println it
    }
else
    println process.text
```

The `Process` class adds a few more methods that serve just as an abbreviation to the existing Java's class methods:

- **`getIn()`**—The abbreviated form of `getInputStream()`
- **`getOut()`**—The abbreviated form of `getOutputStream()`
- **`getErr()`**—The abbreviated form of `getErrorStream()`

One more process method worth noting is the `waitForOrKill()` method, which waits for the process to finish in a given number of milliseconds, or kills it. Listing 4.42 kills the process if it does not finish in one second.

Listing 4.42 Process Handling

```
process = "ls -e".execute()
error = process.err
if (process.waitForOrKill(1000) != 0)
     error.eachLine {
          println it
     }
else
     println process.text
```

This method could be useful because a flaw in the `java.lang.Process` design does not guarantee that the `waitFor()` method will ever end.

Embedding with Java

Now that we have covered the basics of the Groovy programming language and saw how we can use existing Java classes in Groovy scripts, we should focus on Java's side of the story. This section explains how to work with Groovy scripts from Java code. Apache's Bean Scripting Framework (BSF) project and Scripting API (included in JDK 6), general frameworks for embedding various Java-enabled scripting languages, are covered in Chapter 6, "Bean Scripting Framework" and Chapter 9, "Scripting API," respectively. You can use these frameworks for embedding Groovy (and languages covered in Chapter 3) with your application, but Groovy offers a closer integration of scripts and Java code. This section covers this "native" Groovy integration in the Java environment.

Groovy provides the `groovy.lang.GroovyShell` class that represents a shell capable of various operations on scripts. Listing 4.43 provides an example.

Listing 4.43 Evaluating Groovy Scripts from Java

```
import groovy.lang.GroovyShell;

import java.io.IOException;

import org.codehaus.groovy.control.CompilationFailedException;

public class GroovyShellTest {

    public static void main(String[] args) {
        GroovyShell shell = new GroovyShell();
        try {
            Object result = shell.evaluate(
                    "x =1; y =2; return x+y "
            );
            System.out.println(result);
        } catch (CompilationFailedException cfe){
            System.out.println("Syntax not correct " + cfe);
        } catch (IOException ioe){
        }
    }
}
```

As you can see, the `evaluate()` method is used to actually execute the Groovy code and return the result to Java. Two exceptions could be thrown by this method:

- org.codehaus.groovy.control.CompilationFailedException, which indicates errors in the source code compilation
- java.io.IOException, which indicates a generic error during script evaluation

Besides the actual scripting code, the evaluate() method accepts java.io.File and java.io.InputStream parameters, so that you can use it to evaluate scripts contained in files or some other resources. Note that appropriate groovy and asm JAR files have to be in the classpath to successfully run this example.

The groovy.lang.GroovyShell class can also be used to invoke other scripts from Groovy itself:

```
import java.io.File

shell = new GroovyShell()
shell.run(new File("argTest.groovy"), ["arg1", "arg2"])
```

This is not a particularly elegant solution, and one expects to have an easy mechanism for including and evaluating other scripts (such as the mechanisms you can find in most other scripting languages). For that purpose, Groovy defines a special evaluate() command. Look at the following example:

```
import java.io.File

evaluate(new File("argTest.groovy"), ["arg1", "arg2"])
```

This script produces the same result as the preceding example through a more natural mechanism. Besides files, the evaluate() command can also take a string (containing Groovy statements) for an argument.

Usually you want to pass some variables to the script that has to be evaluated. The groovy.lang.Binding class is used for this variable binding task between two environments. To put it simply, this class is used to pass variables in and out of the script's scope, as shown in Listing 4.44.

Listing 4.44 Binding

```
import groovy.lang.Binding;
import groovy.lang.GroovyShell;

import java.io.File;
import java.io.IOException;

import org.codehaus.groovy.control.CompilationFailedException;

public class GroovyShellTest {

    public static void main(String[] args) {
        Binding binding = new Binding();
        binding.setVariable("name", "Groovy");
        binding.setVariable("type", "Scripting");
        GroovyShell shell = new GroovyShell(binding);
        try {
            Object result = shell.evaluate(new File("test.groovy"));
            System.out.println(result);
        } catch (CompilationFailedException e1) {
            System.out.println("Syntax not correct");
        } catch (IOException e3) {
        }
        System.out.println("Platform: "
                          + binding.getVariable("platform"));
    }
}
```

The preceding code makes `name` and `type` variables available in the `test.groovy` script. Also, after the script has been evaluated, it gets the value of the `platform` variable from it. For a `test.groovy` script that looks like this:

```
platform = "All"

"Language description
Name: ${name}
Type: ${type}
"
```

the example Java program prints the following result:

```
Language description
Name: Groovy
Type: Scripting

Platform: All
```

As you can see, the return value of the script is the value of the last line evaluated. As in methods, the `return` keyword is

optional, but you can use it if you want. After the `return` keyword is reached, no other statements are evaluated:

```
if (name == 'Groovy')
    return 'All'
else
    return 'Unix'
println "The End"
```

As I said, all Groovy scripts could be compiled to Java classes with the `groovyc` compiler, and then they could be used in Java programs like standard classes.

Now, let's see how we can use methods and classes defined in Groovy scripts without explicitly compiling them to the bytecode. This approach is useful in the development process, when changes in the source are frequent and you need a rapid development environment.

The `groovy.lang.GroovyClassLoader` class is a specialized class loader that enables Groovy scripts to be loaded as Java classes. Listing 4.45 provides an example of its use.

Listing 4.45 Loading Groovy Scripts as Java Classes

```
import groovy.lang.GroovyClassLoader;
import groovy.lang.GroovyObject;

import java.io.File;

public class EmbeddingTest {

    public static void main(String[] args) {
        GroovyClassLoader loader = new GroovyClassLoader();
        GroovyObject groovyObject = null;
        try {
            Class clazz =
                  loader.parseClass(new File("test.groovy"));
            groovyObject = (GroovyObject) clazz.newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
        Object[] arg = {};

        System.out.println(
             groovyObject.invokeMethod("testMethod", arg)
        );
    }
}
```

As mentioned earlier, every object defined in Groovy implements a `groovy.lang.GroovyObject` interface. This is also true for scripts themselves, so you can use the `parseClass()` method of the `GroovyClassLoader` object to get the script as a class. Then we can make a new instance of that class and call the `invokeMethod()` method to execute any loosely defined method in the script. The code in Listing 4.45 calls the `testMethod()` function (with no arguments) defined in the `test.groovy` script:

```
def testMethod() {
    return "Test"
}
```

And it prints the following text on execution:

```
Test
```

Normally you want to implement the Java interface in Groovy and use that implementation back in the Java application. This approach enables you to have a design-through-interfaces approach in your development, but also enables you to use flexible scripting programming for interface implementation.

Let's start by writing the `IInvoice.java` interface:

```
public interface IInvoice {

    public double getTotal();

    public double getSubtotal();

    public void setSubtotal(double subtotal);

    public void recalculate();

    public void applyTax(int customerid);

}
```

The next step is to write the implementation class in Groovy:

```
import IInvoice

class Invoice implements IInvoice {
```

```
    double subtotal
    double total
    double tax

    Invoice(Double subtotal) {
        this.subtotal = subtotal
    }

    void applyTax(int customerid) {
        if (customerid < 1000)
            tax = 0.2
        else
            tax = 0.3
    }

    void recalculate() {
        total = subtotal * (1 + tax)
    }

}
```

Note that we must define types for all arguments and return values. If we don't, the arguments and return values are assumed to be `java.lang.Object`, which does not comply with our interface. Also, *getter/setter* methods do not have to be explicitly defined because of the GroovyBeans feature (see Listing 4.46).

Listing 4.46 Implementing Java Interfaces in Groovy

```
import groovy.lang.GroovyClassLoader;

import java.io.File;
import java.lang.reflect.Constructor;

public class InvoiceTest {

    public static void main(String[] args) {
        GroovyClassLoader loader = new GroovyClassLoader();
        try {
            Class clazz =
               loader.parseClass(new File("Invoice.groovy"));
            Constructor constr =
               clazz.getConstructor(new Class[] {Double.class});
            IInvoice inv =
               (IInvoice)constr.newInstance(
                   new Object[] {new Double(123.0)}
                );
            inv.applyTax(200);
            inv.recalculate();
            System.out.println(inv.getTotal());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Listing 4.46 is similar to the code used to invoke the standalone method defined in the script. The only difference is that we converted our object to the desired interface (`IInvoice` in this case). After that, the `inv` object can be treated as a regular Java object. After the prototype phase is over, we can compile the script using the `groovyc` compiler and create a new object in standard Java fashion:

```
IInvoice inv = new Invoice()
```

Security

Security issues are commonly intended for two groups of people. The first group comprises users (or system administrators) who want to execute scripts that are not verified and that could compromise system resources. The second group comprises developers who want to provide extension points in their applications. In this case, you probably want to restrict the script's access to certain packages and some system resources.

Groovy is integrated with the Java security model. Because the Java security model is a wide topic, I do not cover it in detail, and you are advised to find more information in outside literature.

As mentioned earlier, there are three ways to execute Groovy scripts: Compile them to Java classes using the `groovyc` compiler, evaluate them from a script file, or evaluate embedded expressions represented as Java strings. When the first approach is used, the generated code can be loaded using one of the existing secure class loaders, and there are no further security considerations.

More often, you will evaluate scripts compiled and loaded during runtime. Let's walk through a simple example of setting security constraints when this approach is used. In Listing 4.47, we restrict scripts evaluated from the Java application and grant them only read permission to the `user.home` system property.

Listing 4.47 Security Example

```
package net.scriptinginjava.ch4;

import groovy.lang.GroovyClassLoader;
import groovy.lang.GroovyCodeSource;
import groovy.lang.GroovyObject;

import java.io.File;


public class SecTest {

    public static void main(String[] args) {
        GroovyClassLoader loader = new GroovyClassLoader();
        GroovyObject groovyObject = null;
        try {
            GroovyCodeSource gcs =
                new GroovyCodeSource(new File("D:/sec.groovy"));
            Class clazz = loader.parseClass(gcs);

            groovyObject = (GroovyObject) clazz.newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }

        Object[] arg = {};
        groovyObject.invokeMethod("run", arg);
}
```

This Java code executes the Groovy script just as we did it earlier. There are no additional security constraints in the code itself. Now imagine the Windows platform, where this class (and the whole Java application) is located on the `C:\` partition. On the other hand, untrusted scripts are located on the `D:\` partition.

We need a simple policy file that defines desired restrictions (for example, `C:\my.policy`). Listing 4.48 provides an example.

Listing 4.48 Security Policy

```
grant codeBase "file:/C:/-" {
 permission java.security.AllPermission;
};

grant codeBase "file:/D:/-" {
 permission java.util.PropertyPermission "user.home", "read";
 permission java.lang.RuntimePermission "accessDeclaredMembers";
};
```

We have granted all permissions to the code executed from the C:\ partition, or rather, to our Java application. Scripts have been granted only the permission to read the user.home system property. We had to grant the accessDeclaredMembers runtime permission because Groovy scripts use reflection heavily. This permission should be granted to all Groovy scripts.

If we now define the malicious sec.groovy script as

```
System.setProperty("user.home", "abc")
println System.getProperty("user.home")
```

and execute the SecTest Java application with the command

```
java chapter5.SecTest
```

the script prints the following output:

```
abc
```

In this case, we didn't define any policy manager or policy file to be used, so no security constraints were applied to the script. But if the application is run with the command line, like this:

```
java -Djava.security.manager \
-Djava.security.policy=C:\my.policy \
chapter5.SecTest
```

you should expect a java.security.AccessControlException to be thrown:

```
java.security.AccessControlException: access denied
(java.util.PropertyPermission user.home write)
. . .
```

To test whether the read permission is valid, just comment the first line of the script:

```
//System.setProperty("user.home", "abc")
println System.getProperty("user.home")
```

Now run the SecTest application again. You should expect output similar to this:

```
C:\Documents and Settings\Dejan
```

When the script is executed from the file, GroovyClassLoader sets the codebase to the file's URL (D:\groovy.sec in this example) so that the appropriate security permission can be applied.

In the last case, scripts are defined as strings directly in the Java source. In this case, GroovyClassLoader cannot automatically determine which codebase to use (see Listing 4.49).

Listing 4.49 Evaluating Inline Scripts Under Security Policy

```
package net.scriptinginjava.ch4;

import groovy.lang.GroovyClassLoader;
import groovy.lang.GroovyCodeSource;
import groovy.lang.GroovyObject;

import java.io.File;


public class SecTest1 {

     public static void main(String[] args) {
          try {
               GroovyClassLoader loader = new GroovyClassLoader();
               String scriptText = "println "
                    + "System.getProperty(\"user.home\")";
               GroovyCodeSource gcs =
                    new GroovyCodeSource(
                       scriptText, "test", "D:/script"
                    );
               Class clazz = loader.parseClass(gcs);
               GroovyObject groovyObject =
                    (GroovyObject) clazz.newInstance();
               Object[] arg = {};
               groovyObject.invokeMethod("run", arg);
          } catch (Exception e) {
               e.printStackTrace();
          }
     }

}
```

The GroovyCodeSource class is used to generate the class from the Groovy source with the specific security policy. With it, we generated a class from the scriptText string variable

named test and run under the D:\script codebase. The codebase does not have to be a valid URL, but we used this one so that the same policy file (my.policy) could be applied. If you run the application with the following command line:

```
java -Djava.security.manager \
-Djava.security.policy=C:\my.policy \
chapter5.SecTest1
```

the application prints out the current user's home directory. You can experiment and change the definition of the scriptText variable to this:

```
String scriptText =
    “println System.setProperty(\”user.home\”, \”abc\”)”;
```

Now, the same AccessControlException is thrown, which means that the code is executed under the expected codebase.

Conclusion

In this chapter, we covered the basic syntax of the Groovy programming language. We saw that this language introduces some important scripting concepts, and its syntax is still close to that of Java. This should guarantee a moderate learning curve for every Java developer.

In Chapter 5, we cover advanced Groovy programming techniques and extensions that could leverage your day-to-day programming tasks.

ADVANCED GROOVY PROGRAMMING

Now that we know how to write and run Groovy scripts, it is time to discuss Groovy extensions and see how we can benefit from them. In Groovy scripts, you can use any Java library and API. However, Groovy offers a few extensions that adapt some of the most popular Java APIs to a scripting programming paradigm. Just as with the classes that handle files and processes, we can find closure support for classes that handle database connections, XML parsing, servlets, and so on.

In this chapter, we cover the following topics:

- How to work with databases in a "Groovy" way (GroovySQL)
- How you can use Groovy to write servlets (GroovyServlet)
- Template processing in Groovy (GroovyTemplate)
- Groovy's support for markup languages (GroovyMarkup)
- How to use Groovy for rapid creation of user interfaces (SwingBuilder)

GroovySQL

You have probably heard about various database tools for Java, such as Object-Relational Mapping (ORM) tools and tools that enable use of the Database Access Object (DAO) design paradigm. Groovy adds different values for working with databases, relying on closures and GStrings.

First, let's briefly summarize database-related concepts in Java. For Java applications, a database is abstracted with a Java Database Connectivity (JDBC) driver. JDBC is the standard Java API for database access, and it guarantees cross-database accessibility in Java. Every database vendor has to provide a suitable JDBC driver for its server. When the driver is initialized, we can create a connection to the database. A connection object is responsible for transaction handling and is used to issue queries to the database. Queries are used to get certain data from the database but also to insert, update, and delete data. All relational databases support Standard Query Language (SQL) as the language used to communicate with databases. SQL is the simple formal language you can use to define data structures and manipulate data. I briefly explain all Java database-related concepts as we use them, which should be enough to help you understand the material in this section. However, if you need more information on this topic, consult the appropriate literature.

SQL support in Groovy is located in the `groovy.sql` package. The most important class found there is `groovy.sql.Sql`. You use this class to handle the crucial part of database-related work, such as connection handling and database querying. If you have modest requirements for database access, this class is all you need to finish the work. Let's go through an example and create a simple database to play with.

Relational databases store data in tables, where each table is a collection of rows, and each row in a table contains the same fields. Our database has only one simple table, `users`, where each row contains data for one specific user (ID, username, and balance in the fields with the appropriate names). You can initialize this example database with the following SQL script (`groovy.sql`):

```
-- Uncomment the following lines if you would like to execute
-- this script in MySQL

--create database groovy;
--
--connect groovy;

create table users (
    id integer not null auto_increment,
    username varchar(128),
    balance float,

    primary key(id)
);

insert into users (username, balance) values ('mike', 250.00);
insert into users (username, balance) values ('joe', 123.50);
```

I wrote the examples in this chapter for the MySQL (www.mysql.com) database server, but any relational database server with the appropriate JDBC driver can be used. If you have the MySQL server installed and you want to use it to run examples from this chapter, type the following on the command line (make sure you uncomment lines that create and connect to the database):

```
$ mysql -u root < groovy.sql
```

After this, the `users` table is created in the `groovy` database. The initialization script inserted two demo users in this database, so we have a table that looks like this:

ID	Username	Balance
1	Mike	250.00
2	Joe	123.50

Now let's write a Groovy script that prints all usernames from this table (see Listing 5.1).

Listing 5.1 GroovySQL Example

```
import groovy.sql.Sql

sql = Sql.newInstance("jdbc:mysql://localhost/groovy", "com.mysql.jdbc.Driver")

sql.eachRow("SELECT * FROM users") {
    println it.username
}
```

For the database we initialized earlier, the preceding script would print the following:

```
mike
joe
```

As you can see, this simple task has a simple solution. First, we used the static `newInstance()`method to create a new `groovy.sql.Sql` object. Two parameters are needed for this task:

- The JDBC URL, which is the platform-independent way of addressing a database. This URL has the following format:

  ```
  jdbc:[subprotocol]:[server]/[databaseName]
  ```

  In our case, the following URL

  ```
  jdbc:mysql://localhost/groovy
  ```

  means that we want to access a MySQL `groovy` database on the local host.
- The class name of the JDBC driver, which has to be present in the classpath.

After that, we called the `eachRow()` method, which takes two arguments: A `String` argument that represents a query to be issued, and a closure that is executed on every row of the result. Database operations cannot be simpler than this, and as we can see, the application developer is not responsible for connection handling anymore because the `Sql` class handles all that work. All you should focus on is the business logic defined in the closure argument.

Now we dig deeper into the `groovy.sql.Sql` class and explore all the possibilities it offers through some examples.

groovy.sql.Sql

As mentioned earlier, `groovy.sql.Sql` is the most important class of the GroovySQL module. In this section, we see how we can initialize this class, use it to issue database queries, and work with some advanced concepts, such as prepared statements, stored procedures, and transactions.

OBJECT CREATION

As Listing 5.1 showed, one way to obtain an instance of the `Sql` object is through the `newInstance()` method. This method has several signatures, and they accept different parameters, allowing developers to use this method according to their needs and habits. Listing 5.2 shows different calls to this method.

Listing 5.2 SQL Object Creation Alternatives

```
import groovy.sql.Sql
import java.util.Properties


sql1 = Sql.newInstance("jdbc:mysql://localhost/groovy"
            , "com.mysql.jdbc.Driver")

sql2 = Sql.newInstance("jdbc:mysql://localhost/groovy", "root", ""
            , "com.mysql.jdbc.Driver")

Properties properties = new Properties()
properties.put("username", "root")
properties.put("password", "")
sql3 = Sql.newInstance("jdbc:mysql://localhost/groovy", properties
            , "com.mysql.jdbc.Driver")
```

The username and password of the database account could be passed to the `newInstance()` method as arguments, or along with other properties using the `java.util.Properties` object.

The JDBC driver class name can also be defined using the `loadDriver()` method, in which case it should not be passed to the `newInstance()` method (see Listing 5.3).

Listing 5.3 The `Sql.loadDriver()` Method

```
import groovy.sql.Sql
import java.util.Properties

Sql.loadDriver("com.mysql.jdbc.Driver")

sql1 = Sql.newInstance("jdbc:mysql://localhost/groovy")

sql1 = Sql.newInstance("jdbc:mysql://localhost/groovy", "root"
            , "")

Properties properties = new Properties()
properties.put("username", "root")
properties.put("password", "")
sql3 = Sql.newInstance("jdbc:mysql://localhost/groovy"
            , properties)
```

In all these cases, the appropriate JDBC driver must be in the classpath.

Another method for obtaining an instance of the `Sql` class is through its constructor. This approach is more suitable in cases when you want to work with existing `java.sql.Connection` objects. A common example is a script that is part of a larger Java application using some kind of connection pool. Database connections are an expensive commodity, and are usually cached in some kind of pool so that they can be shared. The connection is then pulled out of the pool and passed to the script.

Listing 5.4 represents a somewhat simplified scenario where the Java class initiates a connection, calls the Groovy script to do database-related work, and gets the results from it.

Listing 5.4 Initializing the `Sql` Object with a Connection—Java Class

```
import groovy.lang.Binding;
import groovy.lang.GroovyShell;

import java.io.File;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class JDBCExample {

  public static void main(String args[]) {
    Connection con = null;

    try {
      Class.forName("com.mysql.jdbc.Driver").newInstance();
      con = DriverManager.getConnection(
                 "jdbc:mysql://localhost/groovy",
                 "root", ""
      );

      Binding binding = new Binding();
      binding.setVariable("connection", con);
      GroovyShell shell = new GroovyShell(binding);
      String returnValue = (String)shell.evaluate(
                                 new File("jdbc_example.groovy")
                             );
        System.out.println("Return value=" + returnValue);
        System.out.println(binding.getVariable("result"));

    } catch(Exception e) {
      System.err.println("Exception: " + e.getMessage());
    } finally {
         try {
            if(con != null)
              con.close();
```

Listing 5.4 Continued

```
         } catch(SQLException e) {}
      }
   }
}
```

The `jdbc_example.groovy` script could look like that shown in Listing 5.5.

Listing 5.5 Initializing the `Sql` Object with a Connection—Groovy Script

```
import groovy.sql.Sql

if (connection == null) {
    result = null
    return "No valid connection provided"
}

sql = new Sql(connection)
result = "Usernames\n-----\n"
sql.eachRow("SELECT * FROM users") {
    result += "${it.username}\n"
}

return "Success"
```

If we now execute the Java program from Listing 5.4, we should get the following result:

```
Return value=Success
Usernames
-----
mike
joe
```

> **NOTE**
> Note that the returned result indicates the status of the script execution. Actual database data was bound to the `result` variable.

The `DataSource` class was introduced in the JDBC 2.0 Optional Package. Its purpose is to provide an easier and more generic means for obtaining a `Connection` object. This class is often used with the Java Naming and Directory Interface (JNDI), which provides a global memory to store and look up configuration objects. In a typical scenario, an application uses JNDI to look up a `DataSource` object and use that object to create a connection.

If you use JNDI and `DataSources`, you can pass an instance of the `javax.sql.DataSource` class to the `groovy.sql.Sql`

constructor. In this case, the example application from Listing 5.4 could be changed so that it uses a JNDI lookup for a data-source and passes it to the script, which then initiates an `Sql` object with it. The Groovy script could also be used to look up an appropriate `DataSource` object (see Listing 5.6).

Listing 5.6 Creating an `Sql` Object with `DataSource`

```
import javax.sql.DataSource
import groovy.sql.Sql

InitialContext ctx = new InitialContext()

DataSource ds = (DataSource) ctx.lookup("jdbc/MySQLDB")

Sql = new Sql(ds)
```

Database Queries

As you probably know, the SQL language provides an interface between developers and relational databases. A developer selects, inserts, updates, and deletes data in a database by issuing queries written in SQL.

Let's now focus on details of how the `groovy.sql.Sql` class could be used to issue database queries in an easy way.

eachRow. As we saw in our starting example, the `eachRow()` method is used to issue `SELECT` queries to the database. Besides regular strings, the `groovy.lang.GString` object can be used to define queries. GStrings, as we saw in Chapter 4, "Groovy," are used to make it easier to embed Groovy expressions into texts. Listing 5.7 provides an example.

Listing 5.7 The `eachRow()` Method and GString Query Parameters

```
import groovy.sql.Sql
import java.util.Properties

balance = 200

sql = Sql.newInstance("jdbc:mysql://localhost/groovy"
           , "com.mysql.jdbc.Driver")

sql.eachRow("SELECT * FROM users WHERE balance > ${balance}") {
     println it.username
}
```

This modified version of the starting example selects only users with a balance above some threshold value (200 in this case). It prints the following:

```
mike
```

because this is the only user who matches this criterion.

The important thing to learn from this example is that we can embed values directly into the query string.

One more way to embed variables into the query is to use lists, as shown in Listing 5.8.

Listing 5.8 The eachRow() Method and List Query Parameters

```
import groovy.sql.Sql
import java.util.Properties

balance = 200

sql = Sql.newInstance("jdbc:mysql://localhost/groovy"
           , "com.mysql.jdbc.Driver")

params = [100, 200]

sql.eachRow("SELECT * FROM users \
              WHERE balance > ? AND balance < ?"
             , params) {
    println it.username + " " + it[2]
}
```

If a list is passed to the method, each ? character would be replaced with the next element of the list, going from left to right. So this method actually issues the following query:

```
SELECT * FROM users WHERE balance > 100 AND balance < 200
```

As this example shows, you could access columns in the result using both the GroovyBeans syntax (referencing the column name) and the list syntax (referencing the column index). This example prints the following:

```
joe 123.5
```

executeUpdate. To update, delete, or insert database data, the `executeUpdate()` method is used. Now it's time to finish the example from Chapter 4 that handles comma-separated value (CSV) files (see Listing 5.9).

Listing 5.9 The `executeUpdate()` Method and CSV Files

```
import java.io.File
import groovy.sql.Sql

sql = Sql.newInstance("jdbc:mysql://localhost/groovy"
           , "com.mysql.jdbc.Driver")

new File('foo.csv').splitEachLine(',') {
    it.each{
        println "name=${it[0]} balance=${it[1]}"
        result = sql.executeUpdate(
                    "INSERT INTO users (username, balance) \
                     VALUES (${it[0]},${it[1]})"
                  )
        println "${result} line has been added"
    }
}
```

As mentioned earlier, the `splitEachLine()` method would return a list of the lines with another list for the values of each line. Now the `executeUpdate()` method is used to issue the `INSERT` queries to the database. For the `foo.cvs` file, that looks like this:

```
dejan,320.00
alex,640.23
```

The script would issue the following queries:

```
INSERT INTO users (username, balance) VALUES ('dejan', 320.00)
INSERT INTO users (username, balance) VALUES ('alex', 640.23)
```

This method returns a number of modified rows, which are then printed on the console window:

```
name=dejan balance=320.00
1 lines has been added
name=alex balance=640.23
1 lines has been added
```

In this example, we used the `executeUpdate()` method with the `groovy.lang.GString` argument. The same argument signatures that apply for the `eachRow()` method stand here too. Because the `it` closure variable is already a list, we could use more convenient syntax to issue queries:

```
result = sql.executeUpdate(
  "INSERT INTO users (username, balance) values (?, ?)"
  , it
)
```

execute. In situations when you are not sure whether the query is of the SELECT or the INSERT/UPDATE/DELETE type, you should use the `execute()` method, just as with the plain JDBC calls. This method returns `true` when the query has selected some values from the database, and `false` otherwise. See Listing 5.10 for an example.

Listing 5.10 The `execute()` Method

```
import groovy.sql.Sql

sql = Sql.newInstance("jdbc:mysql://localhost/groovy"
          , "com.mysql.jdbc.Driver")

queries = [
"INSERT INTO users (username, balance) values ('mod', 112.0)",
"SELECT * FROM users WHERE username = 'mod'"
]

queries.each {
    result = sql.execute(it)
    if (!result)
        println "${sql.getUpdateCount()} line has been modified"
    else
        println "Selected results"
}
```

As you can see, you can use the `getUpdateCount()` method to retrieve the number of modified rows in the second case, but unfortunately there is no method for obtaining the result set for the SELECT type of queries. So the script prints the following text:

```
1 line has been modified
Selected results
```

This is an unpleasant surprise, which could restrict efficient use of this method. Hopefully, this problem will be corrected soon.

Prepared Statements

When a list is passed to one of these methods, a prepared statement is compiled and executed on the database. In any other case, a regular SQL statement is executed.

Prepared statements represent the ability to compile the SQL statement once and then to use it many times with different parameters. With prepared statements, you place a placeholder (a `?` character) in your SQL statements and then supply the value for it before you execute the statement. With this approach, you can gain both performance and security advantages over plain SQL statements.

Let's see how they can improve the security of our application. Prepared statements separate SQL logic from data. When using the plain SQL statement, you have to be careful to verify the data received from the user because malicious data can change your statement's logic. Take, for example, a case where you have to execute the following query to log in the user:

```
SELECT * FROM users WHERE username = '${username}'
AND password = '${password}'
```

The user supplies the `username` and `password` variables. If he enters the value `admin' --` for the `username` variable and `123` for the `password`, the actual query that would be executed is

```
SELECT * FROM users WHERE username = 'admin'
-- AND password = '123'
```

Because `--` designates the beginning of a comment in SQL, the query becomes simply

```
SELECT * FROM users WHERE username = 'admin'
```

and the user has been granted the administration privileges to the application. You can prevent this behavior with appropriate

argument escaping, but the point is that this kind of attack does not affect the prepared statements because their logic cannot be changed after they have been compiled. This kind of attack is called an SQL injection attack, and using the lists to pass parameters to queries can prevent them from occurring.

STORED PROCEDURES

Stored procedures provide a way to move certain parts of the application logic to the database server. Basically, they are a set of SQL statements powered with basic control and flow statements. Stored procedures are stored on the database server, which means they can improve the performance of tasks that do a lot of database work. Most of the database servers available today support stored procedures, but their syntax and capabilities may vary a great deal. Because of that, you have to be careful how you use them in cases where you want to create a database-independent solution.

The `call()` method is used to execute stored procedures on the database server. The same problem exists as with the `execute()` method—in other words, only the updates can be performed, and there is no way to get the results from the procedure at the moment.

Now let's see one simple use case for stored procedures. This example is written for MySQL Server 5.0 (note that stored procedures in MySQL are not supported for versions prior to this one). Also, stored procedure syntax is different from one vendor to the next, so before you try to execute this example on your database server, consult the server's documentation. To begin, we create a simple stored procedure called `simpleproc` that increases by 10% the balance for all users with a balance above some limit (see Listing 5.11).

Listing 5.11 Stored Procedure

```
delimiter //

CREATE PROCEDURE simpleproc (IN above INT)
BEGIN
    UPDATE users SET balance = balance * 1.1
    WHERE balance > above;
END//
```

The script that actually calls this procedure is similar to all the previous examples in this section (see Listing 5.12).

Listing 5.12 Stored Procedure Call

```
import groovy.sql.Sql
import java.util.Properties

sql = Sql.newInstance("jdbc:mysql://localhost/groovy"
          , "com.mysql.jdbc.Driver")

sql.call("{call simpleproc(?)}", [500])
```

The script calls the procedure with an input parameter value of 500. As with other methods, parameters can be passed using a GString or a list (as in this example).

TRANSACTIONS

Database transactions are an important data management concept. You should use transactions in cases when you want every query from a set of SQL statements to be executed, or when you want none of them to be executed (a property called *atomicity*). When we start a database transaction, no results are reflected to the database data until we *commit* it. If we *roll back* a transaction, results of that transaction's SQL statements execution are dropped.

The `groovy.sql.Sql` class supports transaction handling, but there is one important issue to take care of. JDBC connections are created in the `autocommit` mode by default. That means the transaction is committed after every query, and we have no control over a transaction's behavior. So to use transactions with the `Sql` class, we have to explicitly turn off this feature. Look at Listing 5.13.

Listing 5.13 Transactions

```
import groovy.sql.Sql

sql = Sql.newInstance("jdbc:mysql://localhost/groovy"
          , "com.mysql.jdbc.Driver")
sql.getConnection().setAutoCommit(false)
sql.executeUpdate("INSERT INTO users (username, balance) \
                   VALUES ('Mitch', 212)"
                  )
sql.rollback()
```

In this example, we set the autocommit property of the Connection object to false, executed the query, and rolled back the transaction. As a result, no data is inserted into the database.

groovy.sql.DataSet

This class is an extension of the groovy.sql.Sql class, and it could be interesting to developers who don't want to mess with SQL. It provides the basic interface for handling database tables, which is more than enough for basic operations on a single table. Listing 5.14 provides an example.

Listing 5.14 Introductory DataSet Example

```
import groovy.sql.Sql

sql = Sql.newInstance("jdbc:mysql://localhost/groovy"
          , "com.mysql.jdbc.Driver")

users = sql.dataSet('users')
users.add(username : "Eric", balance: 255.00)
```

Listing 5.14 adds a row in the users table with the values "Eric" and 255 for the username and balance columns, respectively. This example is equivalent to the following Groovy script:

```
import groovy.sql.Sql

sql = Sql.newInstance("jdbc:mysql://localhost/groovy"
         , "com.mysql.jdbc.Driver")

sql.executeUpdate("INSERT INTO users (username, balance) \
                   VALUES ('Eric', 255.00)")
```

However, as we see, the developer does not have to define the SQL query with the groovy.sql.DataSet class. Datasets are definitely an interesting option for junior developers who are not familiar with the SQL language, and for senior developers performing various tasks on a single table (such as saving a map into the table without explicit conversion to the SQL query).

OBJECT CREATION

The `DataSet` object could be created by supplying a `String` parameter to the `dataSet()` method of the `groovy.sql.Sql` class (as we saw in Listing 5.14). Alternatively, the `java.lang.Class` parameter could be passed. If this were the case, the name of the class (without the package name) would be used for table identification. The name of the class is lower-cased before the use. So, if you have defined the `Users` class, for example, you could create a `DataSet` object for the `users` table with the following code snippet:

```
usersClass = new Users()
users = sql.dataSet(usersClass.getClass())
```

Or, with just this:

```
users = sql.dataSet(net.scriptinginjava.ch5.Users)
```

Another approach for creating datasets is to use their constructors. The method signatures are the same as in the earlier examples; just the appropriate `groovy.sql.Sql` object should be passed:

```
users = new DataSet(sql, “users”)
users1 = new DataSet(sql, net.scriptinginjava.ch5.Users)
```

If you want to use this approach, it is often useful to have the `Users` class extend `groovy.sql.DataSet`. That way, you can have the `Users` class defined like this:

```
package net.scriptinginjava.ch5;

import groovy.sql.Sql;

public class Users extends groovy.sql.DataSet {

    public Users(Sql sql) {
        super(sql, Users.class);
    }

}
```

And you can initialize it with the following code:

```
Sql sql = Sql.newInstance(
        “jdbc:mysql://localhost/groovy”
      , “com.mysql.jdbc.Driver”);
Users users = new Users(sql);
```

WORKING WITH DATASETS

In Listing 5.14, we showed how data from the map can be inserted into a database table. A few additional helper methods in this class enable easy data manipulation without using any SQL.

The `each()` method is a substitute for the `eachRow()` method of the `groovy.sql.Sql` class. Listing 5.15 provides an example.

Listing 5.15 The each() Method

```
import groovy.sql.Sql

sql = Sql.newInstance(“jdbc:mysql://localhost/groovy”
           , “com.mysql.jdbc.Driver”)

users = sql.dataSet(‘users’)

users.each() {
    println “${it.username} ${it.balance}”
}
println users.getSql()
```

Because this class operates on one table (the whole table, by default) no queries should be passed. The closure argument would be applied to every row in the table. You can get the actual query that has been executed with the `getSql()` method, which in this case would return the following:

```
select * from users
```

If you need a subset of table data, you can use one of two methods: `createView()` or `findAll()`. These methods do the same thing. Listing 5.16 is an example of the `findAll()` method.

Listing 5.16 The `findAll()` Method

```
import groovy.sql.Sql

sql = Sql.newInstance("jdbc:mysql://localhost/groovy"
           , "com.mysql.jdbc.Driver")

users = sql.dataSet('users')
topUsers = users.findAll{ it.balance > 300 }

topUsers.each() {
    println "${it.username} ${it.balance}"
}

println topUsers.getSql()
```

Listing 5.16 selects all users from the `users` table who have a balance above 300.

As I mentioned, datasets are handy for simple tasks and for developers who are not used to SQL yet.

Groovlets

The Servlet API provides a method for building Web applications in Java. As for other Java technologies presented in this chapter, I do not cover this topic in detail, but give only the basic information needed to introduce you to the concept.

To provide an interaction with the servlets-enabled Web server (servlet container), the Servlet API defines the `javax.servlet.Servlet` interface. You can use implementations of this interface to write Java code executed at users' requests. For that purpose, this interface defines the following method:

```
public void service(ServletRequest request
                  , ServletResponse response)
    throws ServletException, IOException
```

This method is called by a servlet container. The `request` parameter represents an object that contains data specific to a certain HTTP request. On the other hand, the `response` object is used to pass data back to the user.

We have to initialize and register a servlet inside the servlet container, if we want to use it. We do this in the

WEB-INF/web.xml file of the Web application, as we see in a moment.

To run code examples from this section, you need a servlet container. The process of installing the Jetty (http://jetty.mortbay.org) servlet container is described in the "Jetty" sidebar, but you can use any appropriate container of your choice.

JETTY

If you decide to use the Jetty servlet container to run the code examples in this section, you need to install Jetty on your system. To do so, follow these steps:

1. Download Jetty from http://jetty.mortbay.org.
2. Extract the archive in the local directory (for example, `/opt/jetty` on UNIX systems or `C:\jetty` on Windows platforms).
3. Set the `JETTY_HOME` environment variable to point to this directory (for example, `export JETTY_HOME=/opt/jetty`).
4. To run the server, go to the `JETTY_HOME` directory and type `java -jar start.jar`.
5. To stop the server, go to the `JETTY_HOME` directory and type `java -jar stop.jar`.

This demo is a simple Web application that consists of only one Groovy script (Groovlet) and the necessary files used to build and deploy the application (see Figure 5.1).

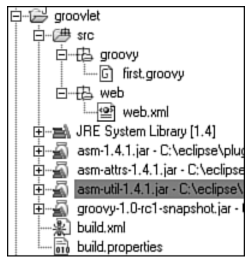


FIGURE 5.1 Web application structure

A specialized servlet that compiles and executes Groovy scripts is located in the `groovy.servlet.GroovyServlet` class. To use this servlet you have to register it in the WEB-INF/web.xml configuration file of your application. Also, you need to ensure all URLs with the .groovy extension (or any

other extension of your choice) are processed by this servlet. The `web.xml` file could look like the one in Listing 5.17.

Listing 5.17 Groovlet Deployment Descriptor

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
 PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
 "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
<servlet>
<servlet-name>Groovy</servlet-name>
<servlet-class>groovy.servlet.GroovyServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>Groovy</servlet-name>
<url-pattern>*.groovy</url-pattern>
</servlet-mapping>
</web-app>
```

In this configuration, the `GroovyServlet` handles all Groovy scripts and compiles them at the time of first execution. The generated bytecode will be cached and executed on every future call. If that script's code changes, it will be recompiled. Because of this compilation process, the appropriate `groovy` and `asm` JAR files must be included in the classpath. You can distribute them with your Web application in the `WEB-INF/lib` directory. On the other hand, if you want to enable Groovlets for every application of your server, just copy them to the appropriate container directory (`JETTY_HOME/ext` in the case of Jetty).

After this, you can put Groovy scripts anywhere around your application, and they will be evaluated.

As with other servlet extensions, Groovlets provide direct mapping of important servlet context variables to scripts, so the following variables are available in them:

- **`out`**—An output writer used to send results back to the client's browser. This variable is actually returned by the `getWriter()` method of the `HttpServletResponse` class.
- **`request`**—The actual `HttpServletRequest` object.

- **session**—Represents a session object from the request. It is an instance of the `HttpSession` class returned by the `getSession(true)` call of the request object from above.
- **application**—An instance of the `javax.servlet.ServletContext` class.

Now, let's write an example script.

```
import java.util.Date

if (session == null) {
  session = request.getSession(true);
}

if (session.counter == null) {
  session.counter = 1
}

println “””
<html>
    <head>
        <title>Groovy Servlet</title>
    </head>
    <body>
Hello, ${request.remoteHost}: ${session.counter}! ${new
Date()}
    </body>
</html>
“””
session.counter = session.counter + 1
```

This example shows how to use the implicit variables mentioned earlier. But more importantly, it shows the real value of Groovy's string enhancements, explained in Chapter 4. You can define the entire HTML content of the page as a string with embedded values. Groovlets and Groovy strings create a powerful environment for generating dynamic Web content, but as we see in a moment, templates have their benefits, and Groovy supports them too.

Now it is time to deploy our application. We use Ant as a build tool and define both `build.properties` and `build.xml` files.

```
build.dir=build
src.dir=src
```

```
# Directory that contains Groovlets
groovy.dir=${src.dir}/groovy

# Directory that contains web.xml
web.dir=${src.dir}/web

# Path to WAR that will be produced
war.file=${build.dir}/${ant.project.name}.war

# Where the WAR should be deployed
webapps.dir=${env.JETTY_HOME}/webapps

# JARs that must be in the WAR
asm.jar=${env.GROOVY_HOME}/lib/asm-1.4.1.jar
groovy.jar=${env.GROOVY_HOME}/lib/groovy-1.0-beta-6.jar
```

The `build.properties` file defines the basic variables needed for the building process, such as the location and name of the Web application archive, the deployment directory, and so on. You need to adapt this file to your local configuration:

```
<project name="groovlet" default="deploy">
    <property environment="env"/>
    <property file="build.properties"/>

    <target name="prepare">
        <mkdir dir="${build.dir}"/>
    </target>

    <target name="war" depends="prepare"
        description="creates WAR file">
        <war destfile="${war.file}"
                      webxml="${web.dir}/web.xml">
            <fileset dir="${groovy.dir}"/>
            <lib file="${groovy.jar}"/>
            <lib file="${asm.jar}"/>
        </war>
    </target>

    <target name="deploy" depends="war"
        description="deploys WAR file">
        <delete dir="${webapps.dir}/${ant.project.name}"/>
        <delete file="${webapps.dir}/${war.file}"/>
        <copy file="${war.file}" todir="${webapps.dir}"/>
    </target>

</project>
```

This XML file contains a few basic tasks usually used for this kind of application. To deploy the project, go to the application's root directory and type the following:

```
ant
```

If everything was set right, the application should be deployed to your server. To test your Groovlet, point your Web browser to the appropriate location, such as:

```
http://localhost:8080/groovlet/first.groovy
```

You should get a response like the one shown in Figure 5.2.

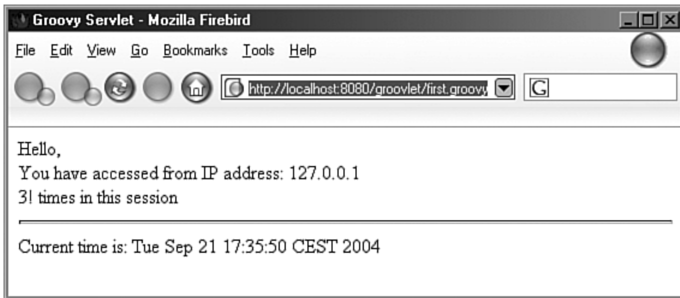


FIGURE 5.2 Groovlet response

Now that we have our application up and running, we can experiment further with it. As I mentioned, Groovy scripts do not have to have the `.groovy` extension, nor can `GroovyServlet` handle only files with that extension. To demonstrate this, we can rename the `first.groovy` script to `first.page` and change the appropriate line in the `web.xml` to the following:

```
<url-pattern>*.page</url-pattern>
```

If we now redeploy our application, the script can be accessed via the following URL:

```
http://localhost:8080/groovlet/first.page
```

In this way, you can hide the technology used to implement the application (Groovy in our case).

Besides the basic context variables mapped directly to the script, all arguments passed with the request are bound to script variables as well. We can modify the example shown in Listing 5.7 that gets all users with a balance above some threshold

value. We provide this script with the ability to be evaluated using the Web browser and return the result formatted as an HTML document (see Listing 5.18).

Listing 5.18 Groovlet Example

```
import groovy.sql.Sql
import java.util.Properties

if (above == null) {
    throw new Exception('above variable must be submitted')
}

sql = Sql.newInstance("jdbc:mysql://localhost/groovy"
          , "com.mysql.jdbc.Driver")

out.println("""
<html>
<head>
<title>Groovy Servlet</title>
</head>
<body>
<table>
EOS)

sql.eachRow("SELECT * FROM users WHERE balance > ${above}") {
    out.println("<tr><td>${it.username}</td></tr>\n")
}

out.println("""
</table>
</body>
</html>
EOS)
```

Here, we check whether the `above` variable has been set, and if it has, the HTML document with the users' information is displayed. So if you go to the following URL:

```
http://localhost:8080/groovlet/balance.page?above=500
```

you receive your desired results (see Figure 5.3).

If you don't submit the `above` parameter, and instead submit something like this:

```
http://localhost:8080/groovlet/balance.page
```

you get an "Internal server error" page (HTTP status code 500) similar to the one shown in Figure 5.4.

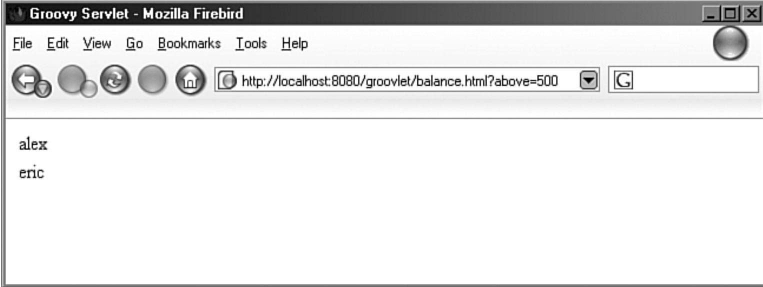


FIGURE 5.3 Groovlet response for valid URL

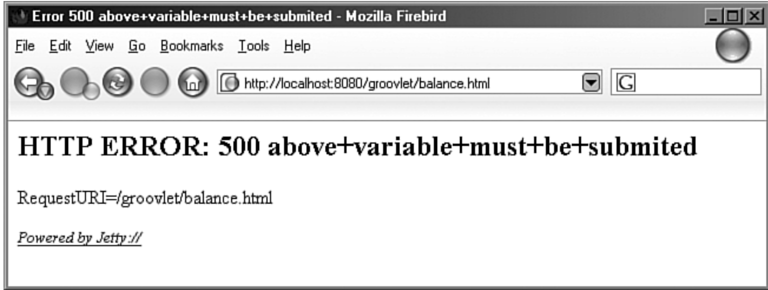


FIGURE 5.4 Groovlet response for invalid URL

This is general `GroovyServlet` behavior, which returns the status code 500 if any exception is thrown during the script evaluation. A status code of 400 is returned if requested script is not found.

We come back to this example in the "GroovyMarkup" section later in this chapter and enable this Groovlet to return an XML result. That way, we create a simple Web service solution with Groovy.

Because many Model/View/Controller (MVC) frameworks are available to Java developers (for example, Spring; go to www.springframework.org for more information) that provide a powerful environment for development of Web applications in Java, I don't recommend that you use Groovlets to develop these kinds of applications from scratch. Instead, use it to leverage your efforts where you find it appropriate.

Groovy Templates

Template engines can ease the task of formatting and reusing large chunks of text. Template engines are often used in the Web development environment. Usually, your Web application has to execute some code, which can retrieve data from the database, for example, and create an HTML document that the Web server returns to the user. To facilitate this task of HTML document creation, Java developers use various template engines, such as JSP and Velocity. These tools allow you to create a file (template) with placeholders where your data is supplied. These placeholders can also contain the code that is executed when the template is evaluated. After the template has been evaluated, the placeholders are substituted with the data values, and the code execution results. That way, we have created a new document from a template.

One additional important use of template engines is the separation of the application's business logic and data presentation. Take the last `balance.page` Groovlet, for example. To begin, we have both aspects here. The code that checks whether needed parameters were submitted and then gets data from the database represents the business logic. But also, this data is formatted using GStrings and is sent back to the browser. A better solution, as we see in a moment, is to create two files: the script that contains only the business logic and the template that formats data supplied by the script. This way, we have a separation of these concerns, and both designers and developers can work on the files they're interested in.

You can find template support for Groovy in the `groovy.text` package. This package contains the abstract `groovy.text.TemplateEngine` class and the `groovy.text.Template` interface. They make it possible to plug any template engine into Groovy (such as Velocity or Freemarker). No matter what engine is used, the API for the Groovy developer remains the same.

Besides that, Groovy offers a template engine implementation located in the `groovy.text.SimpleTemplateEngine` class. This engine provides syntax similar to JSP 2.0 and supports the following expressions for embedding into the template:

- **<% statements %>**—Executes any valid Groovy statements
- **<%= expression %>**—Embeds a Groovy expression into the template
- **${expression}**—The alternative way to embed an expression into a template

Let's now rewrite the balance.page Groovlet with template support (balance_new.page; see Listing 5.19).

Listing 5.19 Groovy Template Support

```
import groovy.sql.Sql
import groovy.text.Template
import groovy.text.TemplateEngine
import groovy.text.SimpleTemplateEngine
import java.io.File

if (above == null) {
    throw new Exception('above variable must be submitted')
}

sql = Sql.newInstance("jdbc:mysql://localhost/groovy"
         , "com.mysql.jdbc.Driver")

users = []

sql.eachRow("SELECT * FROM users WHERE balance > ${above}") {
    users << it.username
}

TemplateEngine engine = new SimpleTemplateEngine()
Template template = engine.createTemplate(
    new File(application.getRealPath("/balance_new.template"))
)
result = template.make(title:"New Balance page", users:users
              , footer:"&copy;Groovy Templates")


result.writeTo(out)
```

As we see, the Groovlet now contains just the application code. It checks whether parameters have been supplied, makes a connection to the database, and gets the results. After this, a template is loaded and evaluated.

The groovy.text.TemplateEngine abstract class has a createTemplate() method that accepts either a File, URL, String, or Reader class instance. In this example, we provided a balance_new.template file located in the root directory of

the Web application. We used the `application` context variable to get the actual path to the file.

In the `groovy.text.Template` interface, there are two signatures for the `make()` method. If you don't pass arguments to it, the `make()` method just evaluates the template and returns the object that implements the `groovy.lang.Writable` interface. You can bind variables to the template by passing a map to the `make()` method. In this example, we provided the title variable (`title`), the list of users (`users`), and the text to be printed as the page footer (`footer`).

The `groovy.lang.Writable` interface defines the `writeTo(java.io.Writer writer)` method used to write itself to the given writer. This interface is used for objects capable of writing themselves to the textual stream, in a more efficient way than just creating their string representation using the `toString()` method.

Now let's look at the `balance_new.template` (see Listing 5.20).

Listing 5.20 Template Example

```
<html>
<head>
<title><%= title %></title>
</head>
<body>
<table>
<%
users.each {
    out.println "<tr><td>${it}</td></tr>"
}
%>
</table>
${footer}
</body>
</html>
```

This template, as is the case with any other template, contains just the data to be printed and expressions defined by the syntax explained earlier.

If you try to execute this Groovlet via the following URL:

```
http://localhost:8080/groovlet/balance_new.page?above=500
```

expect to get a result similar to that generated by the example shown in Listing 5.18. But now, the code is organized better, the

template can be reused among scripts, and junior developers or web designers can change templates without affecting the application's behavior.

One more application for which templates are commonly used is sending HTML-formatted e-mails from standalone applications. I leave you this as an exercise, using techniques you have learned thus far.

GroovyMarkup

We live in a markup world, and markup languages are being used in nearly every Java application. The most popular markup languages used today are XHTML and XML. XML is particularly popular and is used for various tasks from data transfer among applications to data storage in human-readable form.

This capability is crucial to the existence of many Java tools whose purpose is to provide Java programmers with the ability to manipulate markup (especially XML) documents. These tools range from standard XML parsers that implement a Simple API for XML (SAX) or DOM interface, to libraries that enable higher-level mapping between Java objects and XML documents, such as Castor (www.castor.org) and JAXB (http://java.sun.com/xml/jaxb/). These technologies are used heavily, so Groovy tries to make it easy for developers to use them. Groovy introduces a different paradigm from the libraries usually seen in Java applications.

Groovy uses closures and named parameters to create a universal syntax for manipulation of markup data. You then can use this syntax with various builder objects to make a representation of your object's structure in a desired markup format.

Many builder classes are included in Groovy by default, such as the following:

- **`MarkupBuilder`**—Serializes your objects to XML and XHTML
- **`SaxBuilder`**—Can be used with existing SAX handlers
- **`DOMBuilder`**—Creates and parses DOM documents

Also, GroovyMarkup syntax has been proven valuable for manipulation of domain-specific object structures, such as Ant files and Swing user interfaces. For these purposes, you can find the following builder classes in Groovy:

- **AntBuilder**—Used to create Ant build files
- **SwingBuilder**—Used to create Swing user interfaces

In this section, we cover GroovyMarkup syntax and all the builders currently supported. Also, we explain the basic principles of builder classes and write a simple builder.

groovy.xml.MarkupBuilder

This class is used to generate XML or XHTML markup documents. Listing 5.21 generates a simple XML document and prints it to the standard output.

Listing 5.21 Introductory MarkupBuilder Example

```
import groovy.xml.MarkupBuilder

xml = new MarkupBuilder()

xml.users() {
        user(name:'dejanb', balance:200)
        user("mike")
}
```

This script produces the following output:

```
<users>
  <user name='dejanb' balance='200' />
  <user>mike</user>
</users>
```

As you can see, the syntax for generating XML documents is simple. A call to the method of the MarkupBuilder class produces the tag with the same name. If you supply a Map argument to that call, its elements represent arguments of that tag. If an Object argument is passed, it is used as a value for the tag. Theoretically, you could generate tags that have both arguments and values. The child tags are generated in the closure argument of the current tag.

In addition to easy XML generation, the real beauty of this approach is you can have any valid Groovy code inside these closures. To demonstrate this, let's create an example that includes all the Groovy extensions we have learned thus far. Listing 5.22 is an extension of the `balance_new.page` Groovlet (created in Listing 5.19) that can return both XML and XHTML content.

Listing 5.22 Advanced Groovy Programming Example

```
import groovy.sql.Sql
import groovy.text.Template
import groovy.text.TemplateEngine
import groovy.text.SimpleTemplateEngine
import groovy.xml.MarkupBuilder
import java.io.File

if (above == null) {
    throw new Exception('above variable must be submitted')
}

sql = Sql.newInstance("jdbc:mysql://localhost/groovy"
          , "com.mysql.jdbc.Driver")

users = []

sql.eachRow("SELECT * FROM users WHERE balance > ${above}") {
    users << it.username
}

if (type == 'xml') {
    xml = new MarkupBuilder(out)
    xml.users(['above':above]) {
        users.each() {
            xml.user(it)
        }
    }
} else {
    TemplateEngine engine = new SimpleTemplateEngine()
    Template template = engine.createTemplate(new File(
        application.getRealPath("/balance_new.template")
    ))
    result = template.make(title:"New Balance page"
        , users:users, footer:"&copy;Groovy Templates")
    result.writeTo(out)
}
```

Here, we have added just a few lines of code and enabled our Groovlet for XML-via-HTTP Web services. Standard calls to the script result in the same output as before, but if you submit a `type` parameter with the `xml` value, such as

```
http://localhost:8080/groovlet/balance_new.html
?above=500&type=xml
```

an XML document like the following is returned to the user via HTTP. This is a quick and easy way to create a simple Web service.

```
<users above='500'>
  <user>alex</user>
  <user>eric</user>
</users>
```

Note also that no explicit `print` call should be made to these objects. If the object was created with an empty constructor, it would be printed on standard output after the last closure was executed. If you want a different stream to be used, create the object with that stream as the constructor's argument. This was the case with the `xml` object created with the `out` constructor argument in the earlier example.

Because XHTML is sometimes thought of as a sublanguage of XML, there is nothing to stop us from using the `MarkupBuilder` class to create XHTML documents as well. Take a look at Listing 5.23, for example.

Listing 5.23 XHTML Markup Example

```
import groovy.xml.MarkupBuilder

users = ["mike", "joe"]

doc = new MarkupBuilder()
doc.html() {
     head() {
          title("New Balance page")
     }
     body() {
          table() {
               users.each() { user |
                    doc.tr() {
                         doc.td(user)
                    }
               }
          }
     }
}
```

The code in Listing 5.23 produces the same output as the template used in the `balance_new.page` Groovlet. I find templates to be a generally more efficient way to generate HTML because they are more natural and easier to maintain. The

MarkupBuilder class could find its place in the automatic HTML generation of Content Management Systems (CMS).

groovy.util.NodeBuilder

This class is used to create generic treelike structures of arbitrary objects, as shown in Listing 5.24.

Listing 5.24 NodeBuilder Example

```
import groovy.util.NodeBuilder

someBuilder = new NodeBuilder()

root = someBuilder.users([“balance”:100]) {
    user(“mike”)
    user(“joe”)
}

println root
```

As you can see, the syntax for all builders is the same. The difference is that this builder returns an instance of the groovy.util.Node class that represents the root of the created structure. In this example, we printed this node and the result is as follows:

```
users[attributes={balance=100}; value=[user[attributes={};
 value=mike], user[attributes={}; value=joe]]]
```

The Node class provides a few methods you can use to get a certain child node or to iterate through nodes:

```
users = root.get(“user”)

users.each() {
    println “${it.name()} ${it.value()}”
}
```

The get() method returns all children nodes with the given name. In this example, it returns two user nodes defined earlier. With the name() and value() methods, you can obtain the name and value of the given node. So, this code snippet would print the following:

```
user mike
user joe
```

The great thing about nodes is they can also be accessed using pathlike syntax, so you can also write the preceding example in this way:

```
users = root.user

users.each() {
    println "${it.name()} ${it.value()}"
}
```

You can get attributes using the `attributes()` method, which returns a map of attributes. Also, you can use the `attribute(String name)` method to get one specified argument. Another way to obtain an argument is to use the `get()` method and provide the `@` character before the argument's name, as in the following example:

```
root.attributes().each {
    println it
}

println root.attribute("balance")
println root.get("@balance")
```

This code snippet prints the value of the balance attribute of the root node:

```
100
100
```

You can also traverse nodes in various ways. To better understand the methods used for traversing nodes, let's define a more complex tree:

```
builder = new NodeBuilder()

root = builder.users() {
    user(["username":"mike"]) {
        order(["item":"DVD"])
        order(["item":"Book"])
    }
    user(["username":"joe"]) {
        order(["item":"Book"])
        order(["item":"CD"])
    }
}
```

The `depthFirst()` method returns the list of nodes, with child nodes placed before other nodes in the same level. So, the following example:

```
root.depthFirst().each() {
    println "${it.name()} ${it.attributes()}"
}
```

prints

```
users [:]
user [username:mike]
order [item:DVD]
order [item:Book]
user [username:joe]
order [item:Book]
order [item:CD]
```

On the other hand, the `breadthFirst()` method returns the list of nodes, but they are ordered by their level in the hierarchy. So, all nodes of a certain level are placed before nodes of the following level in the hierarchy. The following code snippet:

```
root.breadthFirst().each() {
    println "${it.name()} ${it.attributes()}"
}
```

prints:

```
users [:]
user [username:mike]
user [username:joe]
order [item:DVD]
order [item:Book]
order [item:Book]
order [item:CD]
```

You can also traverse through a node's children using the standard Java iterator:

```
it = root.iterator()
while (it.hasNext()) {
    println it.next()
}
```

groovy.xml.SaxBuilder

SAX is a popular event-based interface for XML parsers. The principle of SAX is simple. To use this API, all you do is

- Write your handler with methods that can be called when a certain event occurs during parsing.
- Register your handler to the parser.
- Start parsing a document.

The `MarkupBuilder` described earlier can create the XML document and print it out to the writer defined in the constructor. The `SaxBuilder` is created to enable the use of existing SAX handlers on XML documents created with GroovyMarkup syntax.

For this demo, we need an XML parser. We could use Xerces (http://xml.apache.org/xerces2-j/), which is Apache's XML parser implementation supporting both SAX and DOM interfaces. Let's create a simple handler for the purpose of the `SaxBuilder` demonstration (see Listing 5.25).

Listing 5.25 SAX Handler Example

```
package net.scriptinginjava.ch5;

import org.xml.sax.Attributes;
import org.xml.sax.helpers.DefaultHandler;

public class SaxHandler extends DefaultHandler {

     public void startDocument() {
          System.out.println("Start document");
     }

     public void endDocument() {
          System.out.println("End document");
     }

     public void startElement(String uri, String name,
               String qName, Attributes atts) {
          if ("".equals(uri))
               System.out.println("Start element: " + qName);
          else
               System.out.println(
                    "Start element: {" + uri + "}" + name
               );
     }

     public void endElement(String uri, String name
             , String qName) {
```

Listing 5.25 Continued

```
        if (“”.equals(uri))
            System.out.println(“End element: “ + qName);
        else
            System.out.println(
                “End element:   {“ + uri + “}” + name
            );
    }

}
```

This handler just prints the note on standard output when a certain event occurs. Events that trigger this handler are the starting and ending tags for the document and elements being parsed.

Now let’s write a script that uses this handler to process our XML document (see Listing 5.26).

Listing 5.26 SaxBuilder

```
import groovy.xml.SAXBuilder
import net.scriptinginjava.ch5.SaxHandler

builder = new SAXBuilder(new SaxHandler())

builder.users([‘above’:500]) {
    user(“Dejan”)
    user(“Mike”)
}
```

This builder is the same as the `MarkupBuilder`, except it’s in the constructor. `SaxBuilder` is initialized with the `org.xml.sax.ContentHandler` interface implementation which is the `SaxHandler` class defined earlier. As a result, this script prints out the following result:

```
Start element: users
Start element: user
End element: user
Start element: user
End element: user
End element: users
```

Note that `SaxBuilder` does not trigger the `startDocument()` and `endDocument()` events.

groovy.xml.DomBuilder

You can also use Groovy's markup syntax with DOM documents. A DomBuilder class enables the parsing and creation of DOM documents. After a document is created, you can use it in a standard manner and even access data using the pathlike syntax. Listing 5.27 provides an example.

Listing 5.27 Parsing Documents with DomBuilder

```
import groovy.xml.DOMBuilder
import java.io.StringReader
import groovy.xml.dom.DOMCategory
import org.apache.xalan.serialize.SerializerToXML

xml = new StringReader("<html><head><title class='mytitle'> \
Test</title></head><body><p class='mystyle'> \
This is a test.</p></body></html>"
)
doc = DOMBuilder.parse(xml)
root = doc.documentElement

elem = DOMCategory.get(root, "head")
SerializerToXML ser = new SerializerToXML()
ser.init(System.out, null)
ser.serialize(elem)

println ""

use (DOMCategory) {
    println root.head
}
```

In this example, we created a new document using the StringReader class. Then we parsed it with the DOMBuilder.parse() method. This parse() method expects the Reader argument so that it can be used to parse files too.

```
xml = new FileReader("test.html")
```

The preceding code line gets data from the test.html file and parses it with the DOMBuilder. After parsing is done, we have an instance of the org.w3c.dom.Document class, which we can use just as we would in Java. The interesting thing in this example is the use of the groovy.xml.dom.DOMCategory class, which is the helper class used for accessing the document's elements. The get() method used earlier is an actual replacement

for the following code snippet, and it is used to find the element with the given name:

```
NodeList nodeList = root.getChildNodes()
for (node in nodeList) {
    if (node instanceof Element) {
        Element child = (Element) node
        child.hasChildNodes()
        if(child.getTagName().equals("head")) {
            elem = child
            break
        }
    }
}
```

Now, we can go one step further and apply the `use()` method (added to the `Object` class in Groovy) and access the nodes with the pathlike syntax. The example shown in Listing 5.27 prints the following:

```
<head><title class="mytitle">Test</title></head>
<head><title class="mytitle">Test</title></head>
```

As you can see, no serializer should be created in the closure passed to the `use()` method to print a node on the display.

Along with parsing existing documents, you can use the `DOMBuilder` markup class to create these documents as well, as shown in Listing 5.28.

Listing 5.28 Creating Documents with `DomBuilder`

```
import groovy.xml.DOMBuilder
import groovy.xml.dom.DOMCategory

builder = DOMBuilder.newInstance()
root = builder.html {
     head {
          title(class:"mytitle", "Test")
     }
     body {
          p (class:"mystyle", "This is a test.")
     }
}

use (DOMCategory) {
    println root.head
}
```

This example creates a document identical to the document parsed in Listing 5.27. Also, it uses the `DOMCategory` class to serialize the `head` node.

groovy.xml.Namespace

Often you want to create XML documents with namespaces to avoid name collisions of elements and attributes. To achieve this in Groovy, use the `groovy.xml.Namespace` builder, as shown in Listing 5.29.

Listing 5.29 Namespace Example

```
import groovy.xml.*
import groovy.xml.dom.*

builder = DOMBuilder.newInstance()
htmlBuilder = new Namespace(
    builder, "http://www.w3.org/TR/REC-html40", "html"
)

root = htmlBuilder.html {
    head {
        title(class:"mytitle", "Test")
    }
    body {
        p (class:"mystyle", "This is a test.")
    }
}

use (DOMCategory) {
    println root
}
```

The `Namespace` builder takes the parent builder, namespace URL, and namespace prefix to be used as the constructor's arguments. In this example, we defined the `Namespace` builder on top of the `DOMBuilder` and created the XHTML document. This script produces the following output:

```
<html:hmtl>
<html:head>
<html:title class="mytitle">Test</html:title>
</html:head>
<html:body>
<html:p class="mystyle">This is a test.</html:p>
</html:body>
</html:hmtl>
```

Each tag now has the prefix (`html:`) added before the tag name.

groovy.util.BuilderSupport

All the builders described thus far extend the `groovy.util.BuilderSupport` abstract class. This class defines a default implementation for most of the methods. It also defines the following abstract methods:

```
protected abstract
 void setParent(Object parent, Object child);
protected abstract
 Object createNode(Object name);
protected abstract
 Object createNode(Object name, Object value);
protected abstract
 Object createNode(Object name, Map attributes);
protected abstract
 Object createNode(Object name, Map attributes
  , Object value);
```

If you want to define a custom builder that enables markup syntax for your object structure, extend this class and implement these methods. Let's now write a simple builder to demonstrate this process (see Listing 5.30).

Listing 5.30 Custom `BuilderSupport` Implementation (`BuilderSupport.groovy`)

```
package net.scriptinginjava.ch5

import groovy.util.BuilderSupport

class CustomBuilder extends BuilderSupport {

    IndentPrinter out

    public CustomBuilder() {
        this.out = new IndentPrinter()
    }

    protected void setParent(Object parent, Object child) {
    }

    protected Object createNode(Object name) {
        out.println "[${name}]"
        return name
    }

    protected Object createNode(Object name, Object value) {
        out.println "[${name} = ${value}]"
        return name
    }

    protected Object createNode(Object name, Map attributes) {
        out.println "[${name} (${attributes})]"
```

Listing 5.30 Continued

```
        return name
    }

    protected Object createNode(Object name, Map attributes
                               , Object value) {
        out.println "[${name} (${attributes}) = ${value}]"
        return name
    }

    public static void main(String[] args) {
        builder = new CustomBuilder()
        builder.html() {
            head() {
                title("CustomBuilder")
            }
            body(class:"bodyStyle") {
                p("Custom builder made this")
            }
        }
    }

}
```

The `BuilderSupport` class has its own implementation of the `invokeMethod()` method, which calls the appropriate `createNode()` method signature. The name of the called method is passed as a `name` argument, and the map is passed as the `attributes` argument. Any other object passed to these methods is mapped to the `value` argument. Based on the number of method arguments, the appropriate signature of the `createNode()` method is called. If closure were passed, it would be used to create child nodes. Knowing this, we can assume our example prints the following result on the console window:

```
[html]
[head]
[title = CustomBuilder]
[body ([class:bodyStyle])]
[p = Custom builder made this]
```

Groovy and Swing

Swing refers to the Java library of GUI controls, such as buttons, check boxes, and so on. This API has been an integral part of the Java 2 platform since its initial release, and it has been

widely used for the creation of graphical user interfaces for Java applications. In Chapter 2, “Appropriate Applications for Scripting Languages,” we discussed issues related to GUI programming. Also, we saw how scripting languages can be used for this task. In this section, we see how the `SwingBuilder` markup class can be used for building Swing user interfaces with Groovy. Also, we compare this technique with traditional Java solutions.

Flexibility of scripting languages, such as Groovy, is important for building user interfaces. For that purpose, Groovy offers the `SwingBuilder` markup class, which enables much faster creation of Swing user interfaces than Java. Let’s start with a simple example (see Listing 5.31).

Listing 5.31 `SwingBuilder`

```
import groovy.swing.SwingBuilder
import javax.swing.*
import java.awt.*

builder = new SwingBuilder()

frame = builder.frame( title:'Update balance', size:[200,100]) {
      panel(layout: new FlowLayout()) {
        label(text:"dejan")
        textField(text:"500", preferredSize:[100,20]
                 , horizontalAlignment:SwingConstants.CENTER)
        button(text:"Update", actionPerformed: { update() })
    }
}

def update() {
    pane = builder.optionPane(
               message:'User data has been updated'
    )
    dialog = pane.createDialog(frame, 'Success')
    dialog.show()
}

frame.show()
```

When evaluated, this script shows a window like the one in Figure 5.5.

FIGURE 5.5 Update Balance window

As you can see from this example, you can create Swing interfaces using the `GroovyMarkup` syntax in just a few lines of code. The `SwingBuilder` class has methods mapped to standard Swing components, so the `frame()` method creates the `javax.swing.Jframe` component, the `panel()` method

javax.swing.JPanel, and so on. These methods accept the Map parameter, which defines the components' attributes. If a closure argument is passed for container components, components defined in the closure are created inside that container. Of course, these closures can contain any valid Groovy code (for example, the code that gets data from the database). The java.awt.event.ActionListener classes are replaced with closures contained in the actionPerformed parameter. In the example, the button component executes the loosely defined update() method when it is clicked.

Comparing Listing 5.31 with the following equivalent code in Java, we can estimate how much time the SwingBuilder class could save for desktop application development:

```
package net.scriptinginjava.ch5;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;


public class gui {

    private static JFrame frame;

    public static void main(String[] args) {
        frame = new JFrame( "Update balance" );
        frame.getContentPane().setLayout( new FlowLayout() );
        frame.setSize( 200, 100 );

        JLabel label = new JLabel("dejan");
        frame.getContentPane().add(label);

        JTextField text = new JTextField();
        text.setPreferredSize( new Dimension(100, 20));
        text.setHorizontalAlignment(SwingConstants.CENTER);
        text.setText("500");
        frame.getContentPane().add(text);

        JButton button = new JButton("Update");
        button.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent av) {
                    gui.update();
                }
            }
        );
        frame.getContentPane().add(button);
        frame.show();
    }

    public static void update() {
```

```
        JOptionPane pane = new JOptionPane(
            “User data has been updated”
        );
        JDialog dialog = pane.createDialog(frame, “Success”);
        dialog.show();
    }

}
```

It is obvious that the Java code is not only twice as long as the Groovy code, but also it is harder to read and maintain.

TableLayout

In addition to the standard Swing components, you can find the `TableLayout` component in the `groovy.swing.impl` package, as shown in Listing 5.32.

Listing 5.32 `TableLayout`

```
import groovy.swing.SwingBuilder
import groovy.sql.Sql

sql = Sql.newInstance(“jdbc:mysql://localhost/groovy”
           , “com.mysql.jdbc.Driver”)
builder = new SwingBuilder()

frame = builder.frame(title:’TableLayout Demo’
             , location:[200,200]
             , size:[300,200]
        ) {
    menuBar {
        menu(text:’Help’) {
            menuItem() {
                action(name:’About’, closure:{ showAbout() })
            }
        }
    }

    tableLayout {
        tr {
            td {
                label(text:”username”)
            }
            td {
                label(text:”balance”)
            }
        }
        sql.eachRow(“SELECT * FROM users \
                    WHERE balance > 500”) { row |
            builder.tr {
                builder.td(colfill:true) {
                    textField(text:row.username)
                }
                builder.td(colfill:true) {
```

Listing 5.32 Continued

```
                    textField(text:
                        (new Float(row.balance)).toString())
                }
            }
        }

        tr {
            td(colspan:2, align:'center') {
                button(text:'OK')
            }
        }
    }
}

frame.show()

def showAbout() {
    pane = builder.optionPane(message:'Scripting in Java demo')
    dialog = pane.createDialog(frame, 'About')
    dialog.show()
}
```

Listing 5.32 results in a window (see Figure 5.6) listing all users with a balance over 500.

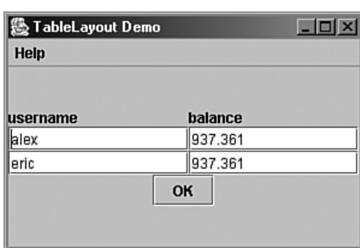


FIGURE 5.6 `TableLayout` Demo window

With the `TableLayout` component, the process of creating Swing user interfaces is practically the same as that for generating HTML pages with `groovy.xml.MarkupBuilder`. It is an interesting option for developers with a Web application background who are more familiar with HTML than with the Swing-programming paradigm.

Listing 5.32 also shows how to create menus for the frame window. The syntax is intuitive, and the only thing that needs explanation is the `action()` method. It accepts the name of the menu item and the closure that is executed when a user clicks a

menu item. In this example, closure just calls the showAbout() loosely coupled method that displays an appropriate dialog.

TableModel

If you have ever tried to use the javax.swing.JTable, you have probably experienced all the flexibility and complexity of this class. The usual approach is to create an implementation of the javax.swing.table.TableModel interface that provides an interface used to obtain data by the JTable class. Groovy offers its own TableModel implementation that handles data. This helper class can ease data manipulation tasks with Swing tables in Groovy. Listing 5.33 provides an example.

Listing 5.33 TableModel

```
import groovy.swing.SwingBuilder
import java.awt.BorderLayout
import javax.swing.BorderFactory
import groovy.sql.Sql

users = []
sql = Sql.newInstance("jdbc:mysql://localhost/groovy"
          , "com.mysql.jdbc.Driver")
sql.eachRow("SELECT * FROM users WHERE balance > 500") {
     users << ['userid' : it.userid, 'username' : it.username
               , 'balance' : it.balance ]
}

updateBalance = {row, value |
     sql.executeUpdate(
          "UPDATE users SET balance = ${value}
          WHERE userid = ${row.userid}"
     )
     row.balance = value
}

builder = new SwingBuilder()

frame = builder.frame(title:'TableModel Demo', location:[200,200]
               , size:[300,200]) {
     panel(layout:new BorderLayout()) {
          scrollPane(constraints:BorderLayout.CENTER) {
               table() {
                    tableModel(list:users) {
                         closureColumn(header:'Id'
                              , read:{row| return row.userid}
                         )
                         closureColumn(header:'Username'
                              , read:{row| return row.username}
                         )
                         closureColumn(header:'Balance'
                              , read:{row| return row.balance}
```

Listing 5.33 Continued

```
                                    , write:updateBalance
                               )
                          }
                     }
                }
           }
      }
frame.show()
```

This example results in a table with users who have a balance over 500 and offers the ability to change the balance (see Figure 5.7).

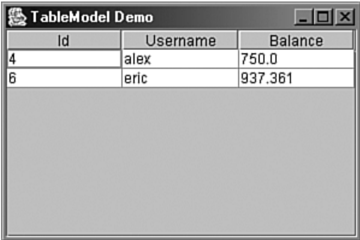


FIGURE 5.7 TableModel example

First, we defined the users list containing a map for each table row. The table model is defined with the tableModel() method within the table() closure. This model is initialized with the previously generated list of users. For each column displayed, a closureColumn() method is defined, which accepts a map parameter with the following data:

- **header**—The column caption.
- **read**—The closure for populating the table. It accepts a Map parameter that represents one member defined in the list (users in this example).
- **write**—An optional closure for updating data in the structure. If the write parameter is not defined, the column is treated as read-only. In this example, we defined only the Balance column with the write parameter, so only this column can be edited. The updateBalance closure has been passed as the value for this parameter.

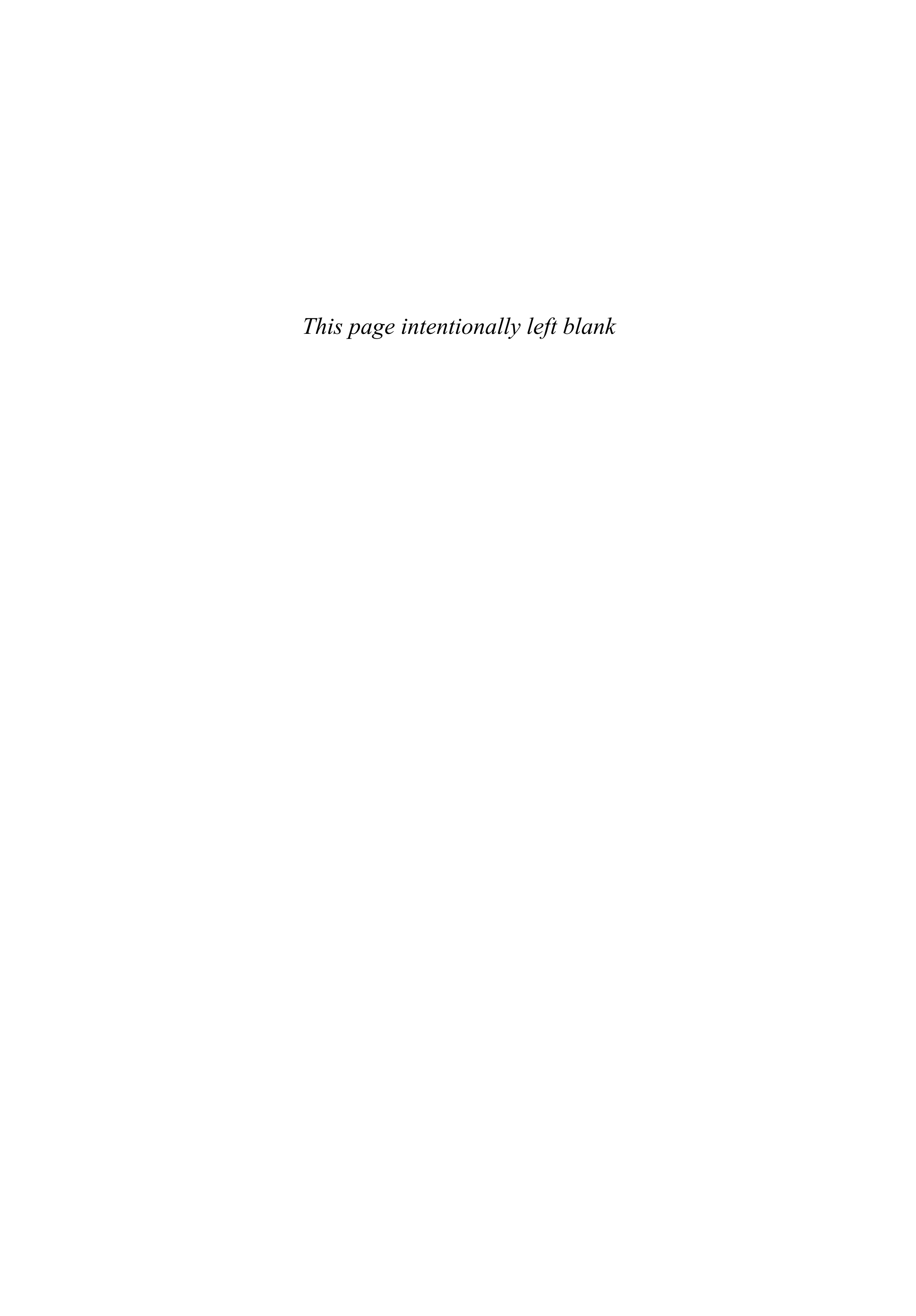
The `write` closure has to accept two parameters. The first is the map that contains old data from the row, and the second is the new value specified for the field. This particular closure updates users' data in the database. Note that because the `TableModel` implements the MVC pattern, the `row` parameter has to be updated also to be displayed properly in the table after the operation.

Conclusion

Thus far, we have covered the basic theory of scripting languages and have seen some of the most important scripting languages available for the Java platform. Also, we have covered the Groovy programming language in detail, including the scripting concepts implemented by this language and the advanced techniques and extensions it brings.

After this introduction in scripting concepts and languages, the following chapters explain where we can apply them in Java projects and what benefits this brings.

But first, in Chapter 6, "Bean Scripting Framework," we dig into another Java library that plays an important role in the story of scripting in Java—the Bean Scripting Framework (BSF), one of the Apache Jakarta Projects. We discuss its purpose and architecture, and look at some code samples.



This page intentionally left blank

Chapter 6

Bean Scripting Framework

In Chapter 2, "Appropriate Applications for Scripting Languages," we discussed the importance of being able to embed scripting languages into your programming environment. In the chapters that followed, we described mechanisms built into the BeanShell, Jython, Rhino, and Groovy scripting languages that enable them to integrate with the Java platform. Although these mechanisms are a natural solution if you want to support only one of these scripting languages, things can get messy if you try to use more than one in your application. Some projects need a general scripting environment in which all languages are treated equally and can be used through the same API. That environment is the Bean Scripting Framework.

Introduction to the Bean Scripting Framework

The Bean Scripting Framework (BSF) is the general Java scripting framework. It is a library, or set of Java classes, that provides a unique API to various script language interpreters (or engines). The BSF supports all languages whose interpreters are implemented in Java. Integration with other, so-called native, engines is also possible through Java Native Interface (JNI) technology.

The BSF provides mechanisms that allow you to evaluate scripts from Java applications. It also serves as an object registry that exposes Java objects for their use in scripts. Java applications can use the BSF to extend or implement some of their functionalities using script languages. This integration with the BSF is language neutral, which means the developer can use any of the supported scripting engines through the same interface. Later, the developer can replace scripted parts of the application with the implementation in another language, without having to modify the Java source code.

In this chapter, we cover the features of the BSF API that you can use in every Java application. After we explain these basic concepts implemented in the BSF, we present some examples of how you can use the framework to extend some existing Java platform technologies. These examples cover two interesting applications for scripting in the Java platform:

- Writing JavaServer Pages (JSP) in languages other than Java
- Using scripting languages for writing extensions that can be used in XSLT transformations

After reading this chapter, you will have a better picture of how to use the BSF in your projects. The examples also show you how to create scripting support for existing Java technologies.

In the following chapters, we use the BSF to demonstrate more advanced scripting techniques and concepts.

Finally, in Chapter 9, "Scripting API," I describe API, which provides similar functionalities as BSF, and as an integral part of JDK 6, successfully succeeds it.

BSF History

The BSF started in 1999 as an IBM research project. The initial goal of this project was to provide access to JavaBeans from scripting languages. Later, BSF development was moved to the IBM AlphaWorks site, which handles IBM's technologies in early development phases (visit www.alphaworks.ibm.com/tech/bsf to learn more).

Soon after, the project was moved to IBM's DeveloperWorks site and became an open source project. Finally, in 2002, IBM donated the BSF to the Apache Software Foundation. At this time, all future development on the BSF will continue as part of Jakarta.

This short history is important because the package name of this project changed during the transition from IBM to ASF. Prior to version 2.3, the package name of the BSF project was `com.ibm.bsf`. From this version on, it is `org.apache.bsf`, so you have to check which version of BSF is supported by the technology (or project) you are working with to use the appropriate distribution.

Getting Started

To start using the BSF, you have to download the `bsf.jar` file and include it in your classpath.

Unfortunately, IBM moved download links from its sites for BSF versions prior to version 2.3. If you need to work with one of these versions, search for the library inside the product that bundles it or some general Java-code repository.

Newer versions, with the new package name (`org.apache.bsf`), are distributed through Apache's Jakarta Web site, http://jakarta.apache.org/bsf/.

Throughout the rest of this chapter, I refer to this newer package name, but all concepts presented are also valid in distributions with the older package name.

Note

From version 2.4 on, BSF uses Jakarta Commons Logging project for its logging needs (http://jakarta.apache.org/commons/logging/). If you use BSF version 2.4 or later, be sure to include an appropriate version of the commons-logging library to be able to run examples from this chapter.

Basic Concepts

In this section, I discuss the basic architecture of the BSF library. I also describe the principles you must follow to work with different scripting languages through the unique programming interface.

Architecture

The `BSFManager` class and the `BSFEngine` interface represent fundamental abstractions in the BSF (see Figure 6.1).

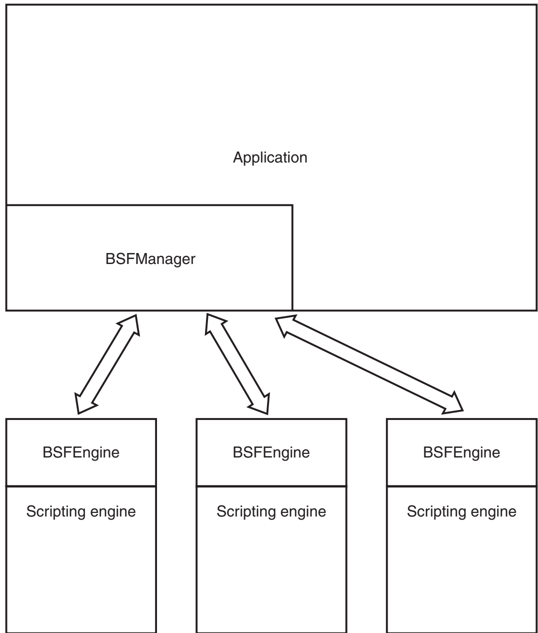


FIGURE 6.1 BSF architecture

The `BSFManager` class is the main feature of this library. It serves as the registry of available scripting engines (interpreters) and is probably the only class that you will use in your Java code.

As we have seen so far, successful integration of certain scripting languages into the Java environment requires an interpreter written in the Java programming language. So naturally,

a general scripting framework such as the BSF should offer a facade for these interpreters. The `BSFEngine` interface answers this need directly. It represents a view to the scripting language interpreter from the Java application. Basic operations on a scripting language interpreter that this interface permits are mapped to its methods. So naturally, every scripting language that wants to comply with the BSF API must implement this interface and map its methods to calls of the appropriate engine implementation.

Registration of Scripting Languages

As we said, the `BSFManager` class represents a central repository of available engines. So to make a certain scripting language accessible through the BSF you have to register it first within the `BSFManager` class. This means you have to register the `BSFEngine` interface implementations because they are engine abstractions in this library.

Many of the scripting languages available for Java are registered by default. For some of them, you can find the implementation of the `BSFEngine` class included within the BSF distribution. The following is a list of scripting languages currently fully integrated with the BSF:

- JavaScript (using Rhino ECMAScript, from the Mozilla project)
- Python (using either Jython or JPython)
- Tcl (using Jacl)
- NetRexx (an extension of the IBM REXX scripting language in Java)
- XSLT Stylesheets (as a component of the Apache XML projects Xalan and Xerces)

Other languages provide their own implementation of the `BSFEngine` interface, but they are still automatically registered to the manager. Those languages are the following:

- Java (using BeanShell, from the BeanShell project)
- JRuby
- JudoScript
- ObjectScript

If you want to use some of these automatically registered languages, all you have to do is to put the appropriate interpreter implementation in the classpath. No additional registration steps are necessary.

Even though the BSF project keeps the list of compatible languages fairly up-to-date, you might find that the language of your choice is not automatically registered. In those cases, you need to register the implementation of the `BSFEngine` interface by yourself. For example, Groovy is a younger project than the BSF, and the BSF didn't support Groovy until the release of BSF version 2.3.0-rc2. So, if you intend to use it with versions prior to this one, you need to register it manually.

You do this through the static `registerScriptingLanguage()` method of the `BSFManager` class:

```
public static void registerScriptingEngine(
    String language
  , String engineClassName
  , String[] extensions
)
```

The following parameters are passed to this method:

- **`language`**—The name of the language. This parameter is used later to associate the scripting engine to scripts of the language that it evaluates.
- **`engineClassName`**—The fully qualified name of the class that implements the `BSFEngine` interface.
- **`extensions`**—An array of file extensions mapped to this language. You can use this helper parameter to find the scripting engine that should process a script file. The `extension` parameter should be `null` if this mapping is not necessary.

Following this method signature (and its description), we can register the Groovy engine using the following code:

```
BSFManager.registerScriptingEngine(
     "groovy"
   , "org.codehaus.groovy.bsf.GroovyEngine"
   , new String[] { "groovy", "gy" }
);
```

With this call, we registered the org.codehaus.groovy.bsf.GroovyEngine class as the language engine for the Groovy language, whose scripts can have groovy or gy extensions.

Manual registration is not always encouraged, especially in situations where you don't want your project to depend on the chosen scripting language or on the BSF distribution that is used—in other words, when you want to have a general-purpose scripting framework.

Unfortunately, the list of registered scripting languages is static and hard-coded, so changing it introduces certain implications. You can find the list of supported languages in the org.apache.bsf.Languages.properties property file located in the BSF JAR file. To change this list, you have to obtain the BSF's source code and rebuild the project. This approach has its drawbacks. One drawback is the fact that maintaining the external project's source increases software management complexity. As such, when you want to upgrade to the next version of the BSF library, these changes could be lost, which could introduce new bugs in your system.

Still, I document this process for those who choose this path anyway. To add the Groovy language to the BSF's list of default-registered languages, you have to locate the Languages.properties file in the org.apache.bsf package of the BSF source code. Next, you have to add the line in that file that defines a desired language. For example, you can register the Groovy programming language by appending the following line in that properties file:

```
groovy = org.codehaus.groovy.bsf.GroovyEngine, groovy|gy
```

As you can see, the syntax is straightforward. The language name serves as the key of the property. In the value, we have the name of the class that implements the BSFEngine interface, and optionally, file extensions that the engine processes. Note that extensions are separated with the | character.

If you included the BSF in your project's source code, your job is done. The next time you rebuild the project, the modified

Language.properties file will be used, and the language will be registered.

If you want to use the BSF packaged as a JAR file, you have to rebuild the project by executing the dist task of the BSF's build.xml file:

```
ant dist
```

This creates the bsf.jar file, located in the dist folder of the BSF source code distribution.

Finally, you probably want to make sure the language is appropriately registered to the manager. To do this, use the static isLanguageRegistered() method of the BSFManager class. The following code snippet demonstrates this method:

```
if (BSFManager.isLanguageRegistered("groovy")) {
    System.out.println(
        "You are free to use Groovy with BSF!"
    );
} else {
    System.out.println(
        "Sorry, you have to register Groovy first!"
    );
}
```

Again, although this method works, you need to be careful if you decide to use this approach to add scripting language support in the BSF. From a maintenance point of view, doing this from your Java application is a cleaner solution.

Manager and Engine Initialization

After we have made sure that all the languages we're interested in are registered to the manager, we are ready to start exploring this library. As Figure 6.1 pointed out, we need an instance of the BSFManager class in our application. Usually there is just one instance of this class in the whole application. The BSFManager is instantiated simply by calling its empty constructor. For instance:

```
BSFManager manager = new BSFManager();
```

While it is initializing, the manager processes the `Languages.properties` file (described earlier) and registers all languages found in it.

So right after this constructor call, we can get the engine of the desired scripting language. To do so, you have to call the `loadScriptingEngine()` method of the manager and pass the name of the language that you want to use:

```
BSFEngine groovyEngine =
  manager.loadScriptingEngine ("groovy");
```

Now that we know the basics of the `BSFManager` class, we are ready to explore the functionalities of the `BSFEngine` interface and see what it can do for us.

Working with Scripts

The basic task that the `BSFEngine` interface provides is the ability to execute scripts. However, this interface allows us to perform this task in two ways. The `BSFEngine` interface enables developers to

- Evaluate a script and get the result of its execution.
- Execute a script (no result is returned to the application).

The BSF library makes a distinction between these two operations. When we expect that the script returns a result of its execution, this process is called *evaluation*. Otherwise, it is called *execution*. Some other libraries and classes that serve the same purpose as `BSFEngine` do not make this distinction between script evaluation and execution. In those libraries, if you are not interested in the result of the evaluation, you can just ignore it. As we see in the coming sections, the same principle can be used in BSF as well.

Let's explore these methods in more detail so that we can see the crucial differences between them.

eval()

The `eval()` method is used to evaluate an expression of some scripting language. It has the following signature:

```
public Object eval(
    String source
  , int lineNo
  , int columnNo
  , Object expression
) throws BSFException;
```

The first three arguments passed to this method are contextual. They provide information about the context in which this expression has to be evaluated. Of course, whether they are used depends on the capabilities of the underlying interpreter. The actual expression is passed as the last `expression` variable. Usually, the expression is of the `String` type, but more flexibility is achieved by declaring it as an `Object`. To understand this more clearly, let's look at the following Java program in Listing 6.1.

Listing 6.1 The `BSFEngine.eval()` Method Example

```
package net.scriptinginjava.ch6;

import org.apache.bsf.BSFEngine;
import org.apache.bsf.BSFException;
import org.apache.bsf.BSFManager;


public class Test {

     public static void main(String[] args) {
          BSFManager.registerScriptingEngine(
                    "groovy",
                    "org.codehaus.groovy.bsf.GroovyEngine",
                    new String[] {"groovy"}
          );
          BSFManager manager = new BSFManager();
          try {
               BSFEngine engine =
                    manager.loadScriptingEngine("groovy");
               Object result =
                    engine.eval("test.groovy", 1, 1, "2+3");
               System.out.println(result);
          } catch (BSFException bsfe) {
               System.out.println(bsfe.getReason());
          }
     }
}
```

In this example, we combined all the previously described steps:

1. We registered the engine of the Groovy scripting language.

2. We created the BSFManager instance.
3. We obtained the Groovy engine abstraction.
4. We evaluated the expression.

As a result of these steps, the example evaluates the simple Groovy expression 2 + 3 and prints the value 5 (of course) on standard output.

You can replace the code marked in bold in Listing 6.1 with a more elegant solution:

```
Object result = manager.eval(
  "groovy", "test.groovy", 1, 1, "2+3"
);
```

The BSFManager class has methods that are practically equivalent to the method definitions found in the BSFEngine interface. The only difference is that the first (extra) parameter, marked as bold in the above code snippet, should be the name of the targeted language so that the manager knows which engine it should use. That is why I said earlier that you will probably use only the manager class in your application. For most of the common uses of this library, the manager class can handle practically all the tasks that you need.

One more thing that you can learn from this example is how to use BSFException. This is the only exception that BSF throws, and it could indicate various problems in script evaluation. The actual reason why the exception is thrown is located in the reason property of this class. It is an integer code, whose meaning you can find in the following BSFException fields:

```
  public static int REASON_INVALID_ARGUMENT = 0
  public static int REASON_IO_ERROR = 10
  public static int REASON_UNKNOWN_LANGUAGE = 20
  public static int REASON_EXECUTION_ERROR = 100
  public static int REASON_UNSUPPORTED_FEATURE = 499
  public static int REASON_OTHER_ERROR = 500
```

The field names are descriptive enough and do not require additional explanation. You can use this error code to refine your error handling further. For example, you can use the following code in the catch block in Listing 6.2.

Listing 6.2 BSF Exception Handling

```
if (bsfe.getReason() == BSFException.REASON_EXECUTION_ERROR) {
      System.out.println("Script execution error");
} else {
      System.out.println("General error");
}
```

Listing 6.2 separates error handing in cases where the script throws an exception from other exceptions that the BSF library could throw (such as an attempt to use an unknown language). This is the most common scenario in `BSFException` handling. Exceptions thrown by the script could be interpreted as application logic exceptions. They will probably be handled differently from exceptions with other error codes that represent infrastructure exceptions in this case.

exec()

The `exec()` method is used to execute a script using the appropriate `BSFEngine` interface implementation. The signature of this method is similar to that for the `eval()` method:

> **Note**
> You can pass script variable values back to the Java application even if you are using the `exec()` method. But you need to use the variable binding mechanism for that task. We discuss this mechanism in the following sections.

```
public void exec(
    String source
  , int lineNo
  , int columnNo
  , Object script
) throws BSFException;
```

Notice that the `exec()` and `eval()` methods are different only in the fact that the `eval()` method returns the result of the script execution. These two method signatures support our discussion of differences between script execution and evaluation that the BSF authors implemented in this library.

Now that you know all these facts, you can expect that the code in Listing 6.3 is equal in functionality to the code from Listing 6.1.

Listing 6.3 The `BSFEngine.exec()` Method Example

```
BSFManager manager = new BSFManager();
try {
    manager.exec("groovy", "test.groovy", 1, 1 ,"println 2+3");
} catch (BSFException bsfe) {
    bsfe.printStackTrace();
}
```

The only difference is that now there is no return value, and the result is printed directly from the script.

To demonstrate further the similarity between these two methods, we dig into the source of the BSF library. The BSF provides the `org.apache.bsf.util.BSFEngineImpl` class that represents a default abstract implementation of the `BSFEngine` interface. Most of the engine implementations for supported languages extend this class and thus share the common functionalities, such as the `exec()` method behavior.

In this default implementation, the `exec()` method is implemented by simply calling the `eval()` method and ignoring the return result:

```
public void exec(
    String source, int lineNo,
    int columnNo, Object script)
throws BSFException {
    eval(source, lineNo, columnNo, script);
}
```

Because this method is not overloaded in most scripting engine implementations, we can conclude that there are no differences (in performance or anything else) between the `eval` and `exec` methods. So, the only question is do you need the result of the script execution (in which case you should use the `eval()` method) or not (when you should use the `exec()` method)? And even if you don't need the result, you can always use the `eval()` method and ignore the return value. By doing so, you will have a cleaner perspective of the API and one less thing to think about.

Working with Script Files

The methods covered in the previous section accept scripts in the form of Java objects, and as we said, they are usually strings. Usually you need to evaluate scripts defined in files. Even though `BSFEngine` does not have methods that allow you to execute a script file directly, some helper classes and methods are available that can help you with this task. Let's go through

an example and define a simple Groovy script (for example, `net/scriptinginjava/ch6/first.groovy`) for starters:

```
println "Hello world"
```

Now let's write the Java application that executes this script file, using the helper methods defined in the BSF API (see Listing 6.4).

Listing 6.4 Executing a Script File

```
package net.scriptinginjava.ch6;

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

import org.apache.bsf.BSFException;
import org.apache.bsf.BSFManager;
import org.apache.bsf.util.IOUtils;

public class FilesTest {

     public static void main(String[] args) {
          BSFManager.registerScriptingEngine(
                    "groovy",
                    "org.codehaus.groovy.bsf.GroovyEngine",
                    new String[] { "groovy", "gy" }
          );
          BSFManager manager = new BSFManager();
          String fileName = "net/scriptinginjava/ch6/first.groovy";
          try {
            String language =
              BSFManager.getLangFromFilename(fileName);
            String script =
              IOUtils.getStringFromReader(
                   new FileReader(fileName)
              );
            manager.exec(language, fileName, 0, 0, script);
          } catch (BSFException bsfe) {
              bsfe.printStackTrace();
          } catch (IOException ioe) {
              ioe.printStackTrace();
          }
     }
}
```

The `org.apache.bsf.util.IOUtils` class is a collection of methods that can help you with the input/output operations. Currently it only contains the `getStringFromReader()` method, which you can use to load the script file into a `String` variable.

Another helper method that we used in this example is the `getLangFromFilename()` static method of the `BSFManager` class. We used this method to determine which language interpreter should be used according to the filename that is passed as an argument. Remember that we supplied file extensions in the engine registration process. Those extensions are mapped to the appropriate engine, and this method uses this information to obtain the appropriate language name. After we have a language name, we can pass it to the `exec()` method.

The program in Listing 6.4 executes our demo script and prints the following to standard output:

```
Hello world
```

It would be better (and cleaner) if the `exec()` and `eval()` methods had signatures that accept `Reader` objects and wrap this code for you, but currently you have to be satisfied with this solution.

Methods and Functions

In addition to executing plain scripts, you can use `BSFEngine` to call functions and methods defined in a script. `BSFEngine` supports two kinds of method calls:

- A call to a *method* or *function* defined in the script. By the term *function*, I mean standalone methods defined directly in the context of the script. By the term *method*, I mean only methods defined within classes.
- A call to an *anonymous function* (or a *closure*, as we referred to it in previous chapters).

Let's take a closer look at these functionalities.

`call()`

This method is used to make a method or a function call:

```
public Object call(Object object, String name
    , Object[] args) throws BSFException;
```

You can use the call() method with both object-oriented and structured programming languages. In object-oriented languages, the object argument determines an object whose method we want to call. If you want to call a standalone method or a function in a language that supports this feature, such as Python, you should pass the null value for this parameter. Arguments of the method call are passed as an array of objects in the args parameter.

Whether a certain language supports function calls, object method calls, or both, depends on the call() method implementation in the appropriate BSFEngine. At the time of this writing, the Jython engine supports function and method calls, the Groovy engine supports only method calls, and the Rhino engine supports only function calls. Be sure to check whether the desired functionality is implemented correctly for languages that you are planning to support before you rely on it.

Let's demonstrate function and method calls with two simple examples. In the first example, we define a Python script that contains a function definition (call.py):

```
def testFunc(name) :
  return "Hello " + name
```

Now, let's call this function from the Java application using the BSF API (see Listing 6.5).

Listing 6.5 The Function Call Example

```
package net.scriptinginjava.ch6;

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

import org.apache.bsf.BSFEngine;
import org.apache.bsf.BSFException;
import org.apache.bsf.BSFManager;
import org.apache.bsf.util.IOUtils;

public class CallTest {

     public static void main(String[] args) {
          BSFManager manager = new BSFManager();
          String fileName = "net/scriptinginjava/ch6/call.py";
          try {
            String language =
              BSFManager.getLangFromFilename(fileName);
            String script = IOUtils.getStringFromReader(
```

Listing 6.5 Continued

```
            new FileReader(fileName)
          );
          BSFEngine engine =
            manager.loadScriptingEngine(language);
          engine.exec(fileName, 0, 0, script);
          Object result = engine.call(
            null, "testFunc", new Object[] {"Dejan"}
          );
          System.out.println(result);
        } catch (BSFException bsfe) {
          bsfe.printStackTrace();
        } catch (IOException ioe) {
          ioe.printStackTrace();
        }
    }

}
```

First, we had to execute the script to make the `testFunc` function available in the interpreter. For that purpose, we used the `exec()` method described earlier. After this, we can use the `call()` method to invoke this function.

When executed, this program prints the value returned from the function:

```
Hello Dejan
```

To demonstrate how to make a method call through the BSF API, we define a class in the Groovy script:

```
class Test {

  def hello(name) {
    return "Hello " + name;
  }

}

return new Test();
```

The `Test` class has only one method, called `hello`, which accepts one parameter. This method is the same in functionality as the previously used Python function.

Notice that the script returns an instance of this class. This is a necessary step because we need an instance of the object on

which we want to execute the method call, because we pass it to the `call()` method call.

The Java program that calls the `hello` method of the `Test` class instance is pretty much the same as that in Listing 6.5 (see Listing 6.6).

Listing 6.6 The Method Call Example

```
package net.scriptinginjava.ch6;

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

import org.apache.bsf.BSFEngine;
import org.apache.bsf.BSFException;
import org.apache.bsf.BSFManager;
import org.apache.bsf.util.IOUtils;

public class CallTest1 {

    public static void main(String[] args) {
        BSFManager.registerScriptingEngine("groovy",
            "org.codehaus.groovy.bsf.GroovyEngine",
            new String[] { "groovy" });
        BSFManager manager = new BSFManager();
        String fileName = "net/scriptinginjava/ch6/obj.groovy";
        try {
            String language =
                BSFManager.getLangFromFilename(fileName);
            String script = IOUtils.getStringFromReader(
                new FileReader(fileName)
            );
            BSFEngine engine =
                manager.loadScriptingEngine(language);
            Object hello = engine.eval(fileName, 0, 0, script);
            Object result = engine.call(hello, "hello",
                new Object[] { "Dejan" });
            System.out.println(result);
        } catch (BSFException bsfe) {
            bsfe.printStackTrace();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }

}
```

In Listing 6.6, we used the `eval()` method to obtain the instance of the `Test` class. That instance was then passed to the `call()` method, along with the method name we want to execute. Of course, we provided the arguments to this method call in the same way we did earlier.

apply()

The apply() method is used to call anonymous functions (closures) in languages that support them. The signature of this method is:

```
public Object apply(
    String source, int lineNo
  , int columnNo, Object funcBody
  , Vector paramNames, Vector arguments
) throws BSFException;
```

This method is not widely supported among scripting engines, and it would probably end with the call to the eval() method (with sufficient parameters being ignored). A default implementation of this method in the BSFEngineImpl class is shown in the following code:

```
public Object apply(
  String source, int lineNo,
  int columnNo, Object funcBody,
  Vector paramNames, Vector arguments)
throws BSFException {
  return eval(source, lineNo, columnNo, funcBody);
}
```

The Groovy BSF engine implementation overrides the apply() method and thus supports the execution of closures using this method.

To demonstrate the apply() method, we create a script that defines a closure and returns it as a result of the execution:

```
hello = {
  return "Hello " + it;
}

return hello;
```

Now we can use the apply() method to call this closure from the Java application (see Listing 6.7).

Listing 6.7 The BSFEngine.apply() Method Example

```
package net.scriptinginjava.ch6;

import java.io.FileReader;
```

Listing 6.7 Continued

```
import java.io.IOException;
import java.util.Vector;

import org.apache.bsf.BSFEngine;
import org.apache.bsf.BSFException;
import org.apache.bsf.BSFManager;
import org.apache.bsf.util.IOUtils;

public class ApplyTest {

    public static void main(String[] args) {
        BSFManager.registerScriptingEngine("groovy",
            "org.codehaus.groovy.bsf.GroovyEngine",
            new String[] { "groovy" });
        BSFManager manager = new BSFManager();
        String fileName = "net/scriptinginjava/ch6/apply.groovy";
        try {
            String language =
                BSFManager.getLangFromFilename(fileName);
            String script = IOUtils.getStringFromReader(
                new FileReader(fileName)
            );
            BSFEngine engine =
                manager.loadScriptingEngine(language);
            Vector arguments = new Vector();
            arguments.add("Dejan");
            Object result = engine.apply(
                fileName, 0, 0, script, null, arguments
            );
            System.out.println(result);
        } catch (BSFException bsfe) {
            bsfe.printStackTrace();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
}
```

If the result of the script evaluation is a closure (as it is in this case), the Groovy engine calls it. Otherwise, it just returns the result of the execution.

Data Binding

So far, we have seen how we can work with scripts of various scripting languages in a uniform way. But to make the methods described earlier useful, we need to provide a context in which scripts are executed. In other words, we need a mechanism that enables us to pass data between the Java application and scripts.

As we saw earlier, all interpreters available for the Java platform provide this functionality, so it is natural to expect that the BSF will provide a uniform API to bind data to various scripting engines.

In addition to being a scripting engine repository, the `BSFManager` class also represents a repository of objects shared between the Java application and scripts. In the following sections, we discuss two mechanisms that you can use to bind data to the scripting engines' context.

Registering Beans

To support this new role, the `BSFManager` class must define methods used to manipulate objects in this repository. These methods are as follows:

- **`void registerBean(String beanName, Object bean)`**—This method puts a bean in the object repository and thus makes it available for scripts and the application.
- **`Object lookupBean(String beanName)`**—This method is used to obtain the previously registered object from the repository. If that bean with the specified name is not located in the repository, a `null` value is returned.
- **`void unregisterBean(String beanName)`**—This method is used to remove the previously registered bean from the repository. If that bean does not exist, this method does not throw any exceptions. It just does nothing.

These methods are used in Java applications to handle data in the object repository. But to have a fully functional mechanism for data sharing, we need to provide the same mechanism for scripts too. For this purpose, the BSF defines the `org.apache.bsf.util.BSFFunction` class. This class provides a subset of the `BSFManager` functionalities. The three methods I just described are among them. An instance of this class is mapped automatically to the `bsf` variable of the scripting engine context, when the scripting engine is loading.

This leads us to conclude that in every script, we can use the `registerBean()`, `lookupBean()`, and `unregisterBean()` methods of the `bsf` variable to manipulate data in the object repository.

In the rest of this section, we implement a simple example that demonstrates this bean registration process. First, we define a `JavaBean` class that we want to share between our Java application and scripts. Next, we create a Java application that registers a bean instance to the BSF object repository. Finally, we see how we can write scripts that can use this object.

To demonstrate the bean registration process, we define a simple JavaBean (see Listing 6.8). Instances of this bean are shared between the Java application and scripts evaluated by it.

Listing 6.8 Register Bean Example—JavaBean Definition

```
package net.scriptinginjava.ch6;

public class Name {

    String firstName;
    String lastName;

    public Name(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

Let's now elaborate on the bean registration mechanism through a simple example. The Java application in Listing 6.9 instantiates a bean and uses the `BeanManager` object to register it to the repository.

Listing 6.9 Register Bean Example—Java Application

```
package net.scriptinginjava.ch6;

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

import org.apache.bsf.BSFException;
import org.apache.bsf.BSFManager;
import org.apache.bsf.util.IOUtils;


public class Bind {

    public static void main(String[] args) {
        BSFManager.registerScriptingEngine(
            "groovy",
            "org.codehaus.groovy.bsf.GroovyEngine",
            new String[] {"groovy"}
        );
        BSFManager manager = new BSFManager();
        try {
            Name name = new Name("Dejan", "Bosanac");
            manager.registerBean("name", name);
            String fileName = "net/scriptinginjava/ch6/bind.groovy";
            String language =
                BSFManager.getLangFromFilename(fileName);
            String script = IOUtils.getStringFromReader(
                new FileReader(fileName)
            );
            manager.exec(language, fileName, 0, 0, script);
            name = (Name)manager.lookupBean("name");
            System.out.println("Hello " + name.getFirstName());
        } catch (BSFException bsfe) {
            bsfe.printStackTrace();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
}
```

In this Java example, we created a bean called `name` and then registered it with the manager (under the same name). As a result, the script evaluated by the manager can access the `name` object. Next, we executed the `bind.groovy` script and printed a line with the new value of the `firstName` property of the `name` bean.

Now take a look at Listing 6.10.

Listing 6.10 Register Bean Example—Script

```
import net.scriptinginjava.ch6.Name

name = bsf.lookupBean("name")
println "Hello " + name.firstName
bsf.registerBean("name", new Name("Mike", "Johnson"))
```

If we create a `bind.groovy` script like the one in Listing 6.10, the Java application prints the following result:

```
Hello Dejan
Hello Mike
```

As you can see, we used the `bsf` script variable to access the bean repository. After the bean is fetched from the repository, we can use it as a regular object created in the script. The example also showed us how to use the `registerBean()` method to make objects created in the script available to the Java application. In this example, we used it to override the already registered bean, but of course, you can use it to register new beans to the repository as well.

This example showed us how we can use the `bsf` script variable to access and modify the object repository from the script. This is an important aspect of the scripting framework because evaluating scripts without the proper context is not useful.

Declaring Beans

The process of registering a bean to a script has one drawback. You have to write the scripts with the `bsf` variable in mind if they need to access objects provided by the Java application. This is not a problem if your scripts are tightly coupled to your application, but if you wanted to use general-purpose scripts, you would probably have to modify them to suit this library.

Luckily, the BSF provides a mechanism that enables us to register Java objects directly to the scripting engine's context. This means that these objects can be used directly in the script, just like the `bsf` variable described earlier. This mechanism is called *bean declaration* and is similar to the principles already explained.

Java objects are declared with the `declareBean()` method, which has just a slightly different signature from the `registerBean()` method. Listing 6.11 is a modified version of Listing 6.10. I first demonstrate the `declareBean()` method here and then explain differences right after.

Listing 6.11 Declare Bean Example—Java Application

```
package net.scriptinginjava.ch6;

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

import org.apache.bsf.BSFException;
import org.apache.bsf.BSFManager;
import org.apache.bsf.util.IOUtils;


public class Declare {

    public static void main(String[] args) {
        BSFManager.registerScriptingEngine(
            "groovy",
            "org.codehaus.groovy.bsf.GroovyEngine",
            new String[] {"groovy"}
        );
        BSFManager manager = new BSFManager();
        try {
            Name name = new Name("Dejan", "Bosanac");
            manager.declareBean("name", name, name.getClass());
            String fileName = "net/scriptinginjava/ch6/bind.groovy";
            String language =
                BSFManager.getLangFromFilename(fileName);
            String script = IOUtils.getStringFromReader(
                new FileReader(fileName)
            );
            manager.exec(language, fileName, 0, 0, script);
            name = (Name)manager.lookupBean("name");
            System.out.println("Hello " + name.getFirstName());
        } catch (BSFException bsfe) {
            bsfe.printStackTrace();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
}
```

As we can see, the `declareBean()` method accepts the third parameter that represents a class of the declared bean. In this example, we passed `name.getClass()`, which gets the class of our bean.

Declared beans are registered to the same object repository, so they can be accessed using the `lookup` method. With this in mind, this program should behave in the same way as our preceding example. The difference is that now we can access the `name` variable directly. So if we change the previously used script as shown in Listing 6.12, it does not raise any errors.

Listing 6.12 Declare Bean Example—Script

```
import net.scriptinginjava.ch6.Name

//name = bsf.lookupBean("name")
println "Hello " + name.firstName
bsf.registerBean("name", new Name("Mike", "Johnson"))
```

As you can see, we commented the `lookupBean()` method call, because the `name` variable is now directly accessible in the script.

Changes in the declared bean's value do not affect the manager's object repository. So to pass the new value back to the Java application, register the bean from the script again.

Compilation

In previous chapters, we saw that scripts in some scripting languages can be compiled directly to the Java bytecode. The BSF does not provide an interface for this direct compilation to the bytecode. Instead, it provides a special class, `org.apache.bsf.util.CodeBuffer`, which you can use to store Java code generated in the "compilation" process. Although this feature is limited in its functionality, I document it here for your reference.

The idea is to create a source code of the Java class that has one service method. This service method, called `exec()` by default, should contain the Java code that is equivalent in functionality to the script that we want to compile.

Three methods serving this purpose are defined in the `BSFManager` class:

- **`compileExpr()`**—Used to compile a script that returns a result
- **`compileScript()`**—Used to compile a script that does not return a result
- **`compileApply()`**—Used to compile an anonymous function

Although these methods are not widely used and supported, I demonstrate how to use them. The common application code for script compilation could look like that shown in Listing 6.13.

Listing 6.13 Compile Example

```
package net.scriptinginjava.ch6;

import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

import org.apache.bsf.BSFException;
import org.apache.bsf.BSFManager;
import org.apache.bsf.util.CodeBuffer;


public class Compile {

    public static void main(String[] args) {

        BSFManager manager = new BSFManager();
        CodeBuffer cb = new CodeBuffer();
        try {
            manager.compileExpr(
                "jython", "hello", 0, 0, "4+5", cb
            );
            cb.setClassName("Hello");
            cb.setPackageName("net.scriptinginjava.ch6");
            FileWriter out =
                new FileWriter(
                    "net/scriptinginjava/ch6/Hello.java"
                );
            PrintWriter pw = new PrintWriter(out);
            cb.print(pw, true);
        } catch (BSFException bsfe) {
            bsfe.printStackTrace();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
}
```

Here, we created a new instance of the `CodeBuffer` class, compiled an expression, and stored the result in it. As you can see, you can use the `CodeBuffer` object to generate the Java source file (in the appropriate package).

But now, things are getting tricky. Let's look at the generated source code in Listing 6.14.

Listing 6.14 Compile Example—Result

```
package net.scriptinginjava.ch6;
public class Hello
{
  org.apache.bsf.BSFManager
    bsf = new org.apache.bsf.BSFManager();

  public java.lang.Object exec()
```

Listing 6.14 Continued

```
    throws org.apache.bsf.BSFException
  {

    return bsf.eval("jython", request.getRequestURI(), 0, 0,
    "4+5"
    );
  }
}
```

A default implementation of the `compileExpr()` method is located in the `org.apache.bsf.util.BSFEngineImpl` class. Almost all languages leave this default implementation, which for some reason tries to use the `request` variable that is related to the servlet environment. This means you cannot use this default implementation in your standalone applications.

We can solve this problem by writing our own engine class and overriding the `compileExpr()` method, as shown in Listing 6.15.

Listing 6.15 Customized `compileExpr()` Method

```
package net.scriptinginjava.ch6;

import org.apache.bsf.BSFException;
import org.apache.bsf.BSFManager;
import org.apache.bsf.engines.jython.JythonEngine;
import org.apache.bsf.util.CodeBuffer;
import org.apache.bsf.util.ObjInfo;
import org.apache.bsf.util.StringUtils;

public class MyEngine extends JythonEngine {

    public void compileExpr(String source, int lineNo
        , int columnNo, Object expr, CodeBuffer cb)
        throws BSFException {

        ObjInfo bsfInfo = cb.getSymbol("bsf");

        if (bsfInfo == null) {
            bsfInfo = new ObjInfo(BSFManager.class, "bsf");
            cb.addFieldDeclaration(
                "org.apache.bsf.BSFManager bsf = "
              + "new org.apache.bsf.BSFManager();");
            cb.putSymbol("bsf", bsfInfo);
        }

        String evalString = bsfInfo.objName
                          + ".eval(\"" + lang + "\", ";
        evalString += "\"" + source + "\", "
                   + lineNo + ", " + columnNo;
        evalString += "," + StringUtils.lineSeparator;
```

Listing 6.15 Continued

```
        evalString += StringUtils.getSafeString(expr.toString())
                    + ")";

        ObjInfo oldRet = cb.getFinalServiceMethodStatement();

        if (oldRet != null && oldRet.isExecutable()) {
            cb.addServiceMethodStatement(oldRet.objName + ";");
        }

        cb.setFinalServiceMethodStatement(
            new ObjInfo(Object.class, evalString)
        );

        cb.addServiceMethodException(
            "org.apache.bsf.BSFException"
        );
    }
}
```

Now we can use this extended Jython engine to compile scripts not bound to the servlet context (see Listing 6.16).

Listing 6.16 Modified Compile Example

```
package net.scriptinginjava.ch6;

import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

import org.apache.bsf.BSFException;
import org.apache.bsf.BSFManager;
import org.apache.bsf.util.CodeBuffer;


public class Compile {

    public static void main(String[] args) {
        BSFManager.registerScriptingEngine(
                "jython",
                "net.scriptinginjava.ch6.MyEngine",
                null
        );

        BSFManager manager = new BSFManager();
        CodeBuffer cb = new CodeBuffer();
        try {
            manager.compileExpr(
                "jython", "hello", 0, 0, "4+5", cb
            );
            cb.setClassName("Hello");
            cb.setPackageName("net.scriptinginjava.ch6");
            FileWriter out =
                new FileWriter(
                    "net/scriptinginjava/ch6/Hello.java"
```

Listing 6.16 Continued

```
                );
            PrintWriter pw = new PrintWriter(out);
            cb.print(pw, true);
        } catch (BSFException bsfe) {
            bsfe.printStackTrace();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
}
```

If we run the previous Java application in Listing 6.16, we get the following Java file, shown in Listing 6.17.

Listing 6.17 Modified Compile Example—Result

```
package net.scriptinginjava.ch6;

public class Hello
{
  org.apache.bsf.BSFManager
    bsf = new org.apache.bsf.BSFManager();

  public java.lang.Object exec()
    throws org.apache.bsf.BSFException
  {

    return bsf.eval("jython", "hello", 0, 0,
    "4+5"
    );
  }
}
```

A simple example application that uses this generated code could look like the code shown in Listing 6.18.

Listing 6.18 Compiled Script Usage Example

```
package net.scriptinginjava.ch6;

import org.apache.bsf.BSFException;


public class CompileTest {

    public static void main(String[] args) {
        Hello hello = new Hello();
        try {
            System.out.println(hello.exec());
        } catch (BSFException bsfe) {
            bsfe.printStackTrace();
        }
    }
}
```

Although we solved the problem, we can see that this default implementation of the `compile()` method is not particularly useful. It just creates a wrapper class around the BSF methods. It would be more useful if some real mapping between scripting and Java code were done, or even better, if the script's compilation to the bytecode was supported.

Applications

A library such as the BSF has many useful applications in projects that want to use scripting languages. I cover some of these use cases in the following chapters. In this section, I describe two interesting applications bundled with the BSF distribution. They help us to learn how to benefit from libraries such as the BSF and give us some ideas of where we can use similar techniques.

This section describes how to use different scripting languages, through the BSF API, together with the JSP and XSLT technologies.

JSP

If you have ever tried to develop a Web application in Java, you are certainly familiar with JavaServer Pages (JSP). JSP is a technology that provides a simplified way to create dynamic Web content in Java Web applications. Its principle is similar to the concepts of PHP and ASP, in that it is designed to enable you to embed dynamic expressions into HTML documents.

There are three main JSP scripting elements: expressions, scriptlets, and tags.

Expressions are used to insert a Java value directly into an HTML document. Their syntax is as follows:

```
<%= java expression %>
```

When a JSP expression like this one is found in a document, its value is evaluated, converted into a string, and embedded into the resulting document. For example, the following code embeds the current time in the document:

```
<%= new java.util.Date() %>
```

Scriptlets, on the other hand, are used to handle more complex programming tasks than just embedding simple values. Their syntax is as follows:

```
<% java code %>
```

Scriptlets can contain any valid Java code. The predefined `out` variable is used to embed values into the document. Look at the following example:

```
<%
 java.util.List users = new java.util.ArrayList();
 users.add("Mike");
 users.add("Joe");
 for (
   java.util.Iterator it = users.iterator();
   it.hasNext();
  ) {
   out.println(it.next() + "<br>");
 }
%>
```

This scriptlet defines a list and then embeds its elements into the document (one element per line).

Finally, *JSP tags* represent the mechanism used for easy encapsulation and reuse of common functionalities. Their syntax is similar to that of XML. For example, we can use the standard JSP `include` tag to include another page in the current document:

```
<jsp:include page="header.jsp"/>
```

Knowing all this, we can create a simple Web application that consists of a single JSP page. First, we need a servlet container in which we can run this example. We can use the same one that we used for the Groovlet example in Chapter 5, "Advanced Groovy Programming."

In this application, we need only a basic `WEB-INF/web.xml` file, which could look like this:

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
 Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <display-name>Scripting in Java</display-name>
</web-app>
```

No special configuration is needed at this point. Next, we create an `index.jsp` file in the root folder of our application. This page includes all the JSP features that we described earlier (see Listing 6.19).

Listing 6.19 Introductory JSP Example

```
<jsp:include page="header.jsp"/>
Hello!<br>
Current time: <%= new java.util.Date() %><br>
Logged in users:<br>
<%
 java.util.List users = new java.util.ArrayList();
 users.add("Mike");
 users.add("Joe");
 for (java.util.Iterator it = users.iterator(); it.hasNext();) {
   out.println(it.next() + "<br>");
 }
%>

<jsp:include page="footer.jsp"/>
```

Also, we need to create the header file for this page:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
    <title>Welcome</title>
</head>

<body>
```

And we need to create the footer file for this page:

```
&copy; Scripting in Java
</body>
</html>
```

If we now deploy this application (under the `bsfjsp` context path) and run it by visiting the appropriate URL, such as http://localhost:8080/bsfjsp/, we can expect that the browser will show us a page similar to the one shown in Figure 6.2.

As we have seen so far, JSP pages provide a way to write expressions and scriptlets using the Java programming language. The BSF project provides JSP tags that enable you to use any of the supported scripting languages for the same task.

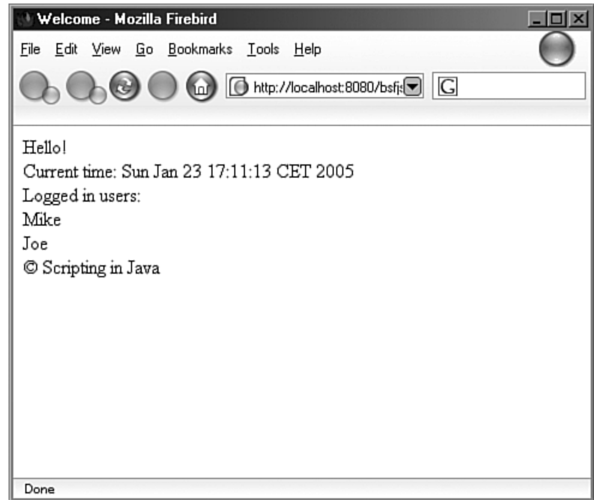


FIGURE 6.2 JSP example

First, we have to set up support for BSF in our Web application. To do this you have to copy the appropriate `bsf.jar` file in the `WEB-INF/lib` folder of your application. Next, we need a Tag Library Descriptor (TLD) file to register desired tags in our application. This descriptor is part of the Jakarta Taglibs project (http://jakarta.apache.org/taglibs/). To run examples from this section, you need to download the Jakarta Taglibs project and copy an appropriate descriptor file to the `WEB-INF/` folder of your Web application (rename it to `bsf.tld` if necessary). To be able to use it, we need to register this tag library. To do that, we modify our `web.xml` file to something like this:

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
 Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <display-name>Scripting in Java</display-name>

  <taglib>
    <taglib-uri>
        http://jakarta.apache.org/taglibs/bsf-2.0
    </taglib-uri>
    <taglib-location>/WEB-INF/bsf.tld</taglib-location>
  </taglib>

</web-app>
```

Now let's write a new JSP page (`index_new.jsp`) that uses Jython to implement the same functionality as our introductory JSP example. Of course, we have to put the appropriate `jython.jar` file in the `WEB-INF/lib` directory as well, if we want to use this interpreter (see Listing 6.20).

Listing 6.20 BSF JSP Example

```
<%@ taglib uri=http://jakarta.apache.org/taglibs/bsf-2.0
 prefix="bsf" %>
<jsp:include page="header.jsp"/>
<bsf:scriptlet language="jython">
from time import *;
</bsf:scriptlet>
Hello!<br>
Current time:
<bsf:expression language="jython">
     strftime('%x %X %Z')
</bsf:expression><br>
Logged in users:<br>
<bsf:scriptlet language="jython">
users = ["Mike", "Joe"]
for user in users:
  out.print(user + "<br>")
</bsf:scriptlet>

<jsp:include page="footer.jsp"/>
```

The difference is that here we used `scriptlet` and `expression` tags defined in the BSF tag library instead of built-in JSP scriptlets and expressions. Both tags require a parameter named `language` that specifies which scripting language is used inside the tag. As you might assume, the difference between these two tags is that the `expression` tag evaluates the script and embeds its return result in the document. The `scriptlet` tag, on the other hand, uses the `out` variable as the writer object to the document.

In this particular example, we first included the BSF tag library defined in the `WEB-INF/web.xml` file. Next, we used the scriptlet to import all functions from Jython's `time` module. The following expression used the `strftime()` function from that module to format the current date and print it in the document. The last expression in the page demonstrates the use of Jython for defining and traversing lists in JSP pages. Note the use of the `out` variable for the purpose explained earlier.

If we now run this new JSP page by visiting http://localhost:8080/bsfjsp/index_new.jsp, we can expect the same result as in the original JSP example.

Xalan-J (XSLT)

The Extensible Stylesheet Language Transformation (XSLT) and XML Path Language (XPath) provide an implementation of the tree-oriented transformation language. You can use them to transform XML documents from one form to another, and to create HTML or text documents from them.

Xalan-J (http://xml.apache.org/xalan-j/) is an Apache XML project that implements the XSLT processor in Java. It's primarily designed for use as the XSLT processor in Java projects, but you can use it as the command-line tool as well.

To use Xalan-J, you have to put the appropriate versions of the `xalan.jar`, `xml-apis.jar` and `xercesImpl.jar` files in your project's classpath.

Let's start this section with a simple example that demonstrates the use of the XSLT and Xalan-J technologies to create an HTML representation of data defined in the XML document. This example is part of the official Xalan-J samples distributed with the library.

First, we have to define an XML document (`namelist.xml`):

```
<?xml version="1.0"?>
<doc>
  <name first="Sanjiva" last="Weerawarana"/>
  <name first="Joseph" last="Kesselman"/>
  <name first="Stephen" last="Auriemma"/>
  <name first="Igor" last="Belakovskiy"/>
  <name first="David" last="Marston"/>
  <name first="David" last="Bertoni"/>
  <name first="Donald" last="Leslie"/>
  <name first="Emily" last="Farmer"/>
  <name first="Myriam" last="Midy"/>
  <name first="Paul" last="Dick"/>
  <name first="Scott" last="Boag"/>
  <name first="Shane" last="Curcuru"/>
  <name first="Marcia" last="Hoffman"/>
  <name first="Noah" last="Mendelsohn"/>
  <name first="Alex" last="Morrow"/>
</doc>
```

We can think of this file as being an XML representation of our customer database. To create an HTML representation of this data, we need to create a simple XSLT transformation (`namelist.xsl`), as shown in Listing 6.21.

Listing 6.21 Simple XSLT Transformation

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xalan="http://xml.apache.org/xalan"
                version="1.0">

  <xsl:template match="/">
    <HTML>
      <H1>XSLT Example</H1>
      <p>
          Here are the names in alphabetical order by last name:
      </p>
      <xsl:for-each select="doc/name">
        <xsl:sort select="@last"/>
        <xsl:sort select="@first"/>
        <p>
        <xsl:value-of select="@last"/>
        <xsl:text>, </xsl:text>
        <xsl:value-of select="@first"/>
        </p>
      </xsl:for-each>
    </HTML>
  </xsl:template>

</xsl:stylesheet>
```

We do not describe XSLT transformations in detail here, because this is not related to the topic of this book. You are advised to consult the appropriate literature on this topic if you need more information about XSLT. Briefly, this transformation traverses all `name` tags under the `doc` tag and creates an HTML document with the `first` and `last` attributes of every such tag.

Now we can apply this transformation to our document. I said that you could use Xalan-J as the command-line processor, so if you type

```
java org.apache.xalan.xslt.Process -IN namelist.xml \
 -XSL namelist.xsl -OUT namelist.html
```

you should find the `namelist.html` document with the following content:

```
<HTML xmlns:xalan="http://xml.apache.org/xalan">
<H1>XSLT Example</H1>
<p>Here are the names in alphabetical order by last name:</p>
<p>Auriemma, Stephen</p>
<p>Belakovskiy, Igor</p>
<p>Bertoni, David</p>
<p>Boag, Scott</p>
<p>Curcuru, Shane</p>
<p>Dick, Paul</p>
<p>Farmer, Emily</p>
<p>Hoffman, Marcia</p>
<p>Kesselman, Joseph</p>
<p>Leslie, Donald</p>
<p>Marston, David</p>
<p>Mendelsohn, Noah</p>
<p>Midy, Myriam</p>
<p>Morrow, Alex</p>
<p>Weerawarana, Sanjiva</p>
</HTML>
```

Of course, Xalan-J (with all its necessary libraries) must be present in your classpath to execute this task successfully.

For those who need more functionality, Xalan-J supports the creation and use of extension elements and functions in the transformation files. We extend the example in Listing 6.21 by implementing a counter that prints the ordering number in front of customers' names.

Let's take a look at the modified transformation file (`javanumlist.xls`) first, shown in Listing 6.22.

Listing 6.22 Extended XSLT Example

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xalan="http://xml.apache.org/xalan"
                xmlns:counter="MyCounter"
                extension-element-prefixes="counter"
                version="1.0">


  <xalan:component prefix="counter"
                   elements="init incr" functions="read">
    <xalan:script lang="javaclass" src="MyCounter"/>
  </xalan:component>

  <xsl:template match="/">
    <HTML>
      <H1>Java Example</H1>
      <counter:init name="index" value="1"/>
      <p>
          Here are the names in alphabetical order by last name:
      </p>
      <xsl:for-each select="doc/name">
```

Listing 6.22 Continued

```
        <xsl:sort select="@last"/>
        <xsl:sort select="@first"/>
        <p>
        <xsl:text>[</xsl:text>
        <xsl:value-of select="counter:read('index')"/>
        <xsl:text>]. </xsl:text>
        <xsl:value-of select="@last"/>
        <xsl:text>, </xsl:text>
        <xsl:value-of select="@first"/>
        </p>
        <counter:incr name="index"/>
      </xsl:for-each>
    </HTML>
  </xsl:template>

</xsl:stylesheet>
```

I added the code snippets marked in bold in Listing 6.22 to register and use the Java class as an XSLT extension. With these fragments, we have registered the extension component with the `counter` prefix and defined that it has two elements (`init` and `incr`) and one function (`read`). Also, we can see that this extension is implemented with the Java class named `MyCounter`.

The `MyCounter` class could look like the one shown in Listing 6.23.

Listing 6.23 Xalan-J Extension Example

```
import java.util.Hashtable;

public class MyCounter {
  static Hashtable counters = new Hashtable ();


  public void init(
      org.apache.xalan.extensions.XSLProcessorContext context
    , org.w3c.dom.Element elem) {
    String name = elem.getAttribute("name");
    String value = elem.getAttribute("value");
    int val;
    try
    {
      val = Integer.parseInt (value);
    }
    catch (NumberFormatException e)
    {
      e.printStackTrace ();
      val = 0;
    }
    counters.put (name, new Integer (val));
  }
```

Listing 6.23 Continued

```
  public int read(String name)
  {
    Integer cval = (Integer)counters.get(name);
    return (cval == null) ? 0 : cval.intValue();
  }

  public void incr(
      org.apache.xalan.extensions.XSLProcessorContext context
    , org.w3c.dom.Element elem) {
    String name = elem.getAttribute("name");
    Integer cval = (Integer) counters.get(name);
    int nval = (cval == null) ? 0 : (cval.intValue () + 1);
    counters.put (name, new Integer (nval));
  }
}
```

As you can see, this class is responsible for manipulation of multiple counters, which are accessed by their names. Extension functions and elements used in the transformation are defined as methods of this class.

Elements must follow a strictly defined method signature. They accept the `org.apache.xalan.extensions.XSLProcessorContext` class instance as the first parameter and the `org.w3c.dom.Element` object as the second parameter. The `Element` parameter contains values passed as the attributes of the element tag in the transformation file. In our example, the `init` and `incr` elements are used to initialize and increment the counter. The particular counter to be used is defined by the `name` parameter passed to the methods.

Functions, on the other hand, can have any signature, and as you can see, they can return the value embedded into the document. The syntax for calling the function is, of course, different from the one used for elements. Listing 6.23 defines the `read` function that is used to return the current value of the counter of our interest.

After this analysis, we can conclude that the preceding transformation first initializes a counter named `index` to the value of `1` by calling the `init` element. Next, in each iteration, it prints its current value (using the `read` function) and increments it (with the `incr` element).

The result of applying this transformation to the XML document defined earlier:

```
java org.apache.xalan.xslt.Process -IN namelist.xml \
-XSL javanamelist.xsl -OUT javanamelist.html
```

results in an HTML document that is slightly different from the one created in the first example (`javanamelist.html`):

```
<HTML xmlns:xalan="http://xml.apache.org/xalan">
<H1>JavaScript Example.</H1>
<p>Here are the names in alphabetical order by last name:</p>
<p>[1]. Auriemma, Stephen</p>
<p>[2]. Belakovskiy, Igor</p>
<p>[3]. Bertoni, David</p>
<p>[4]. Boag, Scott</p>
<p>[5]. Curcuru, Shane</p>
<p>[6]. Dick, Paul</p>
<p>[7]. Farmer, Emily</p>
<p>[8]. Hoffman, Marcia</p>
<p>[9]. Kesselman, Joseph</p>
<p>[10]. Leslie, Donald</p>
<p>[11]. Marston, David</p>
<p>[12]. Mendelsohn, Noah</p>
<p>[13]. Midy, Myriam</p>
<p>[14]. Morrow, Alex</p>
<p>[15]. Weerawarana, Sanjiva</p>
</HTML>
```

As you can see, we used the Xalan-J extension mechanism to add ordering numbers in front of customer names.

Here we have seen how to use Java to extend basic XSLT functionality. Some may find that writing a new Java class for these small extensions is inflexible and overwhelming. Scripting snippets, on the other hand, could be a good fit for this task. You can integrate them easily into XSLT documents because the transformation files are also kept in plain text form. The Xalan-J project provides the ability to embed scripts into the transformation files through its integration with the BSF library. In the following example, we see how to use JavaScript snippets to write transformation extensions better suited to this purpose.

The transformation shown in Listing 6.24 is equivalent in functionality with Listing 6.22 (`javascriptnumlist.xsl`).

Listing 6.24 Xalan-J BSF Example

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                 xmlns:xalan="http://xml.apache.org/xalan"
                 xmlns:counter="MyCounter"
```

Listing 6.24 Continued

```
                extension-element-prefixes="counter"
                version="1.0">

  <xalan:component prefix="counter"
                   elements="init incr" functions="read">
    <xalan:script lang="javascript">
      var counters = new Array();

      function init (xslproc, elem) {
        name = elem.getAttribute ("name");
        value = parseInt(elem.getAttribute ("value"));
        counters[name] = value;
        return null;
      }

      function read (name) {
        return "" + (counters[name]);
      }

      function incr (xslproc, elem)
      {
        name = elem.getAttribute ("name");
        counters[name]++;
        return null;
      }
    </xalan:script>
  </xalan:component>

  <xsl:template match="/">
    <HTML>
      <H1>JavaScript Example.</H1>
      <counter:init name="index" value="1"/>
      <p>
          Here are the names in alphabetical order by last name:
      </p>
      <xsl:for-each select="doc/name">
        <xsl:sort select="@last"/>
        <xsl:sort select="@first"/>
        <p>
        <xsl:text>[</xsl:text>
        <xsl:value-of select="counter:read('index')"/>
        <xsl:text>]. </xsl:text>
        <xsl:value-of select="@last"/>
        <xsl:text>, </xsl:text>
        <xsl:value-of select="@first"/>
        </p>
        <counter:incr name="index"/>
      </xsl:for-each>
    </HTML>
  </xsl:template>

</xsl:stylesheet>
```

The only difference is the fact that, in this example, the entire extended functionality is embedded directly into the XSLT document. I used JavaScript here, but we could also use any of other languages supported by the BSF. With this approach, no separate Java class is needed, and both functions and elements are defined as script functions directly within the transformation file. The same restriction for the method signatures of defined elements applies here too.

ISSUES

Although this method works, you should take care of a few additional considerations if you are planning to use the BSF with Xalan-J.

First, due to historical reasons, Xalan-J still uses the old package name for BSF classes (`com.ibm.bsf`). So it is best to use the BSF library distributed with Xalan-J (you can find it in the `bin` directory).

Next, version 2.2 of the Xalan-J project is bundled with version 1.4 of the Sun JDK. That version of Xalan-J does not support the script extensions we have just described. To use the newer version of Xalan-J with JDK 1.4, prepend the appropriate JAR files to the boot classpath. One way to achieve this is to use the `-Xbootclasspath/p` switch of the `java` command:

```
java \
-Xbootclasspath/p:C:\xalan-j\bin\xalan.jar;\
C:\xalan-j\bin\xml-apis.jar;C:\xalan-j\bin\xercesImpl.jar \
org.apache.xalan.xslt.Process -IN namelist.xml \
-XSL jsnamelist.xsl -OUT jsnamelist.html
```

Also worth noting is the fact that version 1_5R5 of the Rhino interpreter has a bug that throws an exception if you try to use it with Xalan-J. If you experience this problem, try to use it with some other version of Rhino.

Conclusion

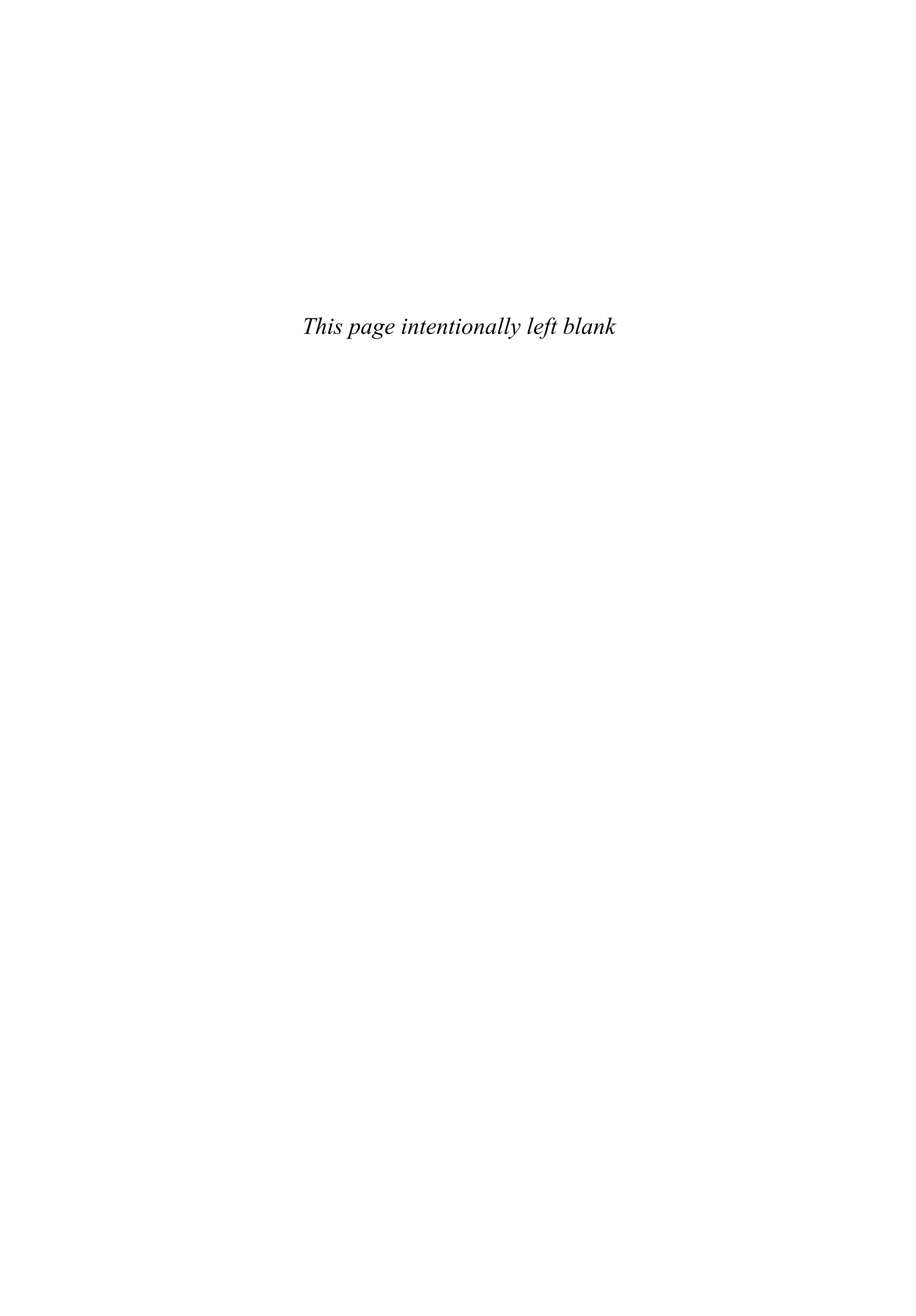
In this chapter, we covered the BSF library architecture, which provides a unique interface to various scripting engines. We learned how we could use it in our Java applications and saw some of its use cases. In the following chapters, we discuss some successful deployments of the BSF library for various tasks related to Java development.

But first, in Chapter 7, "Practical Scripting in Java," we focus on applying the technologies and concepts that we have learned thus far. We dig through some practical tasks that we could simplify by using scripting languages appropriately.

PART III

CHAPTER 7 Practical Scripting in Java

CHAPTER 8 Scripting Patterns


This page intentionally left blank

Chapter 7

Practical Scripting in Java

Earlier in the book, we discussed the roles that scripting languages played (and still do play) in various systems. In this chapter, we see how scripting languages and technologies covered in the previous chapters can be used in those roles. This chapter does not focus on best practices for extending and implementing application functionalities with scripting languages; we cover that, along with scripting patterns and the use of scripting languages in system architecture, in Chapter 8, "Scripting Patterns." Instead, in this chapter, we focus on using scripting languages for tasks that are a part of every development process. Specifically, we see the benefits (and limitations) of writing unit tests in Groovy. Along with unit testing, this chapter focuses on using scripting languages for the following applications:

- Interactive debugging
- Writing Ant build files
- UNIX shell scripting and creating application startup scripts
- Administration and management of Java (and other) systems

Unit Testing

In today's era of extreme programming and unit testing, developers spend a lot of time writing test cases. In fact, you could consider unit testing one of the main tasks directly related to the success of a software project. Unit testing also consumes a lot of development time, so we need to think about how we can perform this task both quickly and satisfactorily.

Using scripting languages to write unit tests brings multiple benefits, all of which reduce the amount of time needed to gain the same advantages as we do with Java-written test cases. Powerful data structures, for example, provide an easy way to define test data. Consider the native support for lists and maps, easy manipulation of regular expressions, and other aspects of Groovy and how it can help you test your application.

A main drawback of the decision to use scripting languages for testing purposes is that it takes more time to execute such tests. This is related to the general characteristics of scripting and programming languages, as we saw in Chapter 1, "Introduction to Scripting." But because test execution is usually not a time-critical operation, we can conclude that scripting languages can be a better choice than system-programming ones in many cases. Many organizations, for example, use a continuous integration technique that suggests batch execution of tests many times a day. Martin Fowler and Matthew Foemmel, well-known software experts, explained it this way (www.martinfowler.com/articles/continuousIntegration.html):

An important part of any software development process is getting reliable builds of the software. Despite its importance, we are often surprised when this isn't done. We stress a fully automated and reproducible build, including testing, that runs many times a day. This allows each developer to integrate daily, thus reducing integration problems.

If we operate in an environment like that which Fowler and Foemmel describe, it is obvious that the amount of time it takes to execute tests is not that crucial. Still, we would like to improve our development speed by writing those tests faster.

Many modern Java IDEs do a great job of helping developers be efficient when unit testing, so it may look like there are no additional benefits to gain from using scripting languages in this field.

But as with other concepts that we describe in the remainder of this book, there is no need to be strict when choosing scripting over the system-programming approach. These technologies play well together, and it is up to you to decide which one to use for your particular task.

Another issue related to the acceptance of scripting languages for Java code testing is the fact that Java developers have a lot of experience with current Java unit-testing tools, such as JUnit. But it is important to note that concepts that we present in this chapter do not collide with the basic principles built into JUnit. As mentioned, you can think of the scripting concepts that I am going to talk about as just a natural extension to JUnit that introduces greater flexibility where you need it. Of course, performance penalties must be paid, so it is up to you to decide whether scripting is beneficial in your environment.

Groovy is an excellent tool for unit testing of Java code. It has the Java syntax, it has all the benefits of a scripting language, and it provides a natural integration with JUnit (the unit-testing framework for Java; see www.junit.org for more information). In the rest of this section, I recap the basics of JUnit and then discuss how Groovy can fit into your unit-testing process. To run examples from this section, you need to download JUnit and include it in the classpath.

JUnit Basics

To begin, let's summarize the basic principles used in JUnit. You write unit tests by extending the `junit.framework.TestCase` class. You use the `setUp()` method of this class to prepare the necessary resources for the test (for example, to open a network connection). Accordingly, you should use the `tearDown()` method to clean up used resources (to close the network connection opened by the `setUp()` method). These two methods are called before and after tests are executed. Actual tests are located in methods whose names start with the `test` keyword.

The `TestCase` class provides a few methods that you can use to verify that test variables have expected values and to mark the current test as failed.

Listing 7.1 demonstrates one simple test case.

Listing 7.1 JUnit Example

```
package net.scriptinginjava.ch7;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import junit.framework.TestCase;

public class MyTest extends TestCase {

     List testList;

     public void setUp() {
          testList = new ArrayList();
          testList.add(new Integer(1));
          testList.add(new Integer(3));
          testList.add(new Integer(5));
          testList.add(new Integer(7));
     }

     public void testInit() {
       assertEquals(4, testList.size());
     }

     public void testIteration() {
          for (Iterator it = testList.iterator(); it.hasNext();) {
              if (((Integer)it.next()).intValue() > 5) {
                   fail("Element greater than 5 found!");
              }
          }
     }

     public static void main(String[] args) {
          junit.textui.TestRunner.run(MyTest.class);
     }
}
```

In this example, the `setUp()` method is used to create a list of four elements. The actual test methods (`testInit()` and `testIteration()`) are used to check whether this list has four elements as expected and whether there are elements greater than 5.

Here you can see the use of the `assertEquals()` method, whose purpose is to verify that two values are identical. Also, we used the `fail()` method to mark the second test as failed if

a certain condition is met. In our example, the `testIteration` test will fail because we defined the element with a value of 7 in the test list.

To run this test case without defining a test suite (we discuss test suites in a moment), we need to define the `main()` method for our test case class. The `main()` method calls the static `run()` method of the `junit.textui.TestRunner` class. The class of the test case is provided as an argument to this method.

NOTE
JUnit also provides other test runner classes that you can use to view results in a graphical user interface, for example, but this textual one is the most appropriate for displaying results in this book.

Now we can run the test case with the following command (JUnit must be in the classpath):

```
java net.scriptinginjava.ch7.MyTest
```

If you want to avoid the `main()` method in the test class, you can execute the test case with the following command:

```
java junit.textui.TestRunner net.scriptinginjava.ch7.MyTest
```

In both cases, we can expect the following output:

```
..F
Time: 0.02
There was 1 failure:
1) testIteration(chapter7.MyTest)
junit.framework.AssertionFailedError:
Element greater than 5 found!
  at net.scriptinginjava.ch7.MyTest.testIteration(
    MyTest.java:28
  )
  at sun.reflect.NativeMethodAccessorImpl.invoke0(
    Native Method
  )
  at sun.reflect.NativeMethodAccessorImpl.invoke(
    Unknown Source
  )
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(
    Unknown Source
  )
  at net.scriptinginjava.ch7.MyTest.main(
    MyTest.java:34
  )

FAILURES!!!
Tests run: 2,  Failures: 1,  Errors: 0
```

As mentioned, the `testIteration` test will fail when it reaches the element with a value of 7.

The GroovyTestCase Class

Now let's see how we can write an equivalent test in Groovy, as Groovy comes with a `groovy.util.GroovyTestCase` class that extends JUnit's standard `TestCase` class. This class provides Groovy integration with JUnit and adds more functionality to the original `TestCase` class.

Let's write a test case that is equivalent to the earlier one and see `GroovyTestCase` in action (see Listing 7.2).

Listing 7.2 `GroovyTestCase` Example

```
package net.scriptinginjava.ch7;

public class MyGroovyTest extends GroovyTestCase {

     List testList;

     public void setUp() {
          testList = [1,3,5,7]
     }

     public void testInit() {
       assertEquals(4, testList.size());
     }

     public void testIteration() {
          if (testList.any {it > 5}) {
               fail("Element greater than 5 found!");
          }
     }
}
```

The first thing you will notice is that this solution is much shorter and simpler than the original test case written in Java. We can see that the same functionality provided by JUnit can be used in these tests too. In addition, we can now use closures, native support for lists, and all the other features that Groovy has to offer.

You might also notice that we didn't define the `main()` method for this class. The `GroovyTestCase` class handles that task, so running these tests is as simple as running any other Groovy script:

```
groovy net/scriptinginjava/ch7/MyGroovyTest.groovy
```

The result that we get this time is similar to the one produced by the pure Java solution:

```
..F
Time: 0.13
There was 1 failure:
1) testIteration(net.scriptinginjava.ch7.MyGroovyTest)
junit.framework.AssertionFailedError:
Element greater than 5 found!
  at sun.reflect.NativeMethodAccessorImpl.invoke0(
    Native Method
  )
  at sun.reflect.NativeMethodAccessorImpl.invoke(
    NativeMethodAccessorImpl.java:39
  )

...

org.codehaus.classworlds.Launcher.launchStandard(
    Launcher.java:410
  )
  at org.codehaus.classworlds.Launcher.launch(
    Launcher.java:344
  )
  at org.codehaus.classworlds.Launcher.main(
    Launcher.java:461
  )

FAILURES!!!
Tests run: 2,  Failures: 1,  Errors: 0
```

The important thing to notice is the execution time, which even for this simple example is considerably longer than that for the previous Java example. But this is the cost of flexibility and faster implementation. It is up to you to estimate the impact of both factors on your project and to decide the most appropriate method of unit testing in your application.

Assertion Methods

In our examples thus far, we used the `assertEquals()` method to check whether objects have expected values. In addition to this method, JUnit provides a few similar useful methods:

- **`assertFalse()`**—Asserts that supplied condition is `false`
- **`assertTrue()`**—Asserts that supplied condition is `true`
- **`assertNull()`**—Asserts that supplied object is `null`
- **`assertNotNull()`**—Asserts that supplied object is not `null`

- **assertSame()**—Asserts that two supplied objects refer to the same object
- **assertNotSame()**—Asserts that two supplied objects do not refer to the same object

In addition to these regular assertion methods, the GroovyTestCase class provides a few more methods that could be valuable in certain situations. Most of them are intended for testing Java arrays. In the following subsections, we discuss these methods and provide examples of how they're used.

assertLength()

You can use the assertLength() method to assert that an array has the expected length, as shown in Listing 7.3.

Listing 7.3 A GroovyTestCase.assertLength() Method Example

```
void testLength() {
    assertLength(15, “ScriptingInJava”.toCharArray())
}
```

This method is applicable to arrays of chars (char[]), primitive integers (int[]), and objects (Object[]).

assertArrayEquals()

As its name implies, this method is used to assert that two arrays are equal (see Listing 7.4).

Listing 7.4 A GroovyTestCase.assertArrayEquals() Method Example

```
void testArray() {
    array1 = new Integer[] {1,3,5,7}
    array2 = new Integer[] {1,3,5,7}
    assertArrayEquals(array1, array2)
}
```

You can use it only on arrays of objects (Objects[]). This method simply calls the assertEquals() JUnit method on every corresponding index element of two arrays.

assertContains()

The assertContains() method is used to assert that a certain element is in the array (see Listing 7.5).

Listing 7.5 A GroovyTestCase.assertContains() Method Example

```
void testContains() {
    char c = 'J';
    assertContains(c, "ScriptingInJava".toCharArray());
}
```

It is applicable to arrays of char and int types (char[] and int[], respectively).

assertToString()

This method asserts that the value of the toString() method call on the given object matches the given text string (see Listing 7.6).

Listing 7.6 A GroovyTestCase.assertToString() Method Example

```
void testToString() {
    f = new java.io.File("test.groovy");
    assertToString(f, "test.groovy");
}
```

assertScript()

The assertScript() method asserts that the given script runs without any exceptions (see Listing 7.7).

Listing 7.7 A GroovyTestCase.assertScript() Method Example

```
void testScript() {
    assertScript("println 3");
}
```

shouldFail()

This method is somewhat different from the preceding methods. It asserts that a given closure throws an exception when it is evaluated (see Listing 7.8).

Listing 7.8 A `GroovyTestCase.shouldFail()` Method Example

```
void testFail() {
    cl = {
        println 3
        throw new java.io.IOException()
    }
    shouldFail(cl)
}
```

You can also call this method with two parameters. When you call it in this way, its first parameter should be the class of the exception that is expected:

```
void testFailIO() {
    cl = {
        println 3
        throw new java.io.IOException()
    }
    shouldFail(java.io.IOException, cl)
}
```

These methods can come in handy in certain situations. You can play with the examples provided here to see how they behave in different contexts.

Test Suites

The unit-testing methods that we have covered to this point work if all the tests are located in one class (script). However, many times our tests are spread out in different classes for different modules.

To run all these tests at once, we need a `TestSuite` class that contains a collection of tests to run. Listing 7.9 is an example of a test suite that contains all the tests from the `MyTest` class defined at the beginning of this section.

Listing 7.9 `TestSuite` Example

```
package net.scriptinginjava.ch7;

import junit.framework.Test;
import junit.framework.TestSuite;
import junit.textui.TestRunner;

public class MySuite {
```

Listing 7.9 Continued

```
    public static Test suite() {

        TestSuite suite = new TestSuite();
        suite.addTestSuite(MyTest.class);

        // add more tests here

        return suite;
    }

    public static void main(String[] args) {
       TestRunner.run(suite());
    }
}
```

As you can see, an instance of `junit.framework.TestSuite` can be passed to the `run()` method of the `TestRunner` class, at which point, all tests defined in that suite will be executed.

Because we defined the `main()` method in this class, we can execute it with the following command:

```
java net.scriptinginjava.ch7.MySuite
```

When executed, the `TestRunner` class searches for the appropriate `suite()` method and executes all the tests defined in it. So we can execute the suite defined earlier with the following command as well:

```
java junit.textui.TestRunner \
net.scriptinginjava.ch7.MySuite
```

The result of this suite's execution is equivalent to the execution of the single test case defined in it.

If you wrote your tests partially in Java and partially in Groovy, you cannot use the standard `TestSuite` class to run all of them. Groovy provides the `groovy.util.GroovyTestSuite` class that you can use to run test cases written both in Java and in Groovy (see Listing 7.10).

Listing 7.10 `GroovyTestSuite` Example

```
package net.scriptinginjava.ch7;

import groovy.util.GroovyTestSuite;
```

Listing 7.10 Continued

```
import junit.framework.Test;
import junit.textui.TestRunner;

public class MyGroovySuite {

     public static Test suite() throws Exception {

          GroovyTestSuite suite = new GroovyTestSuite();
          suite.addTestSuite(MyTest.class);
          suite.addTestSuite(
            suite.compile(
              "net/scriptinginjava/ch7/MyGroovyTest.groovy"
            )
          );

          // add more tests here

          return suite;
     }

     public static void main(String[] args) throws Exception {
        TestRunner.run(suite());
     }
}
```

The only difference from the example in Listing 7.9 is that now we have defined our suite as an instance of the `GroovyTestSuite` class, and we added tests defined in the Groovy script. We used the `compile()` helper method of this class to load the script as a class.

All other issues related to the `junit.framework.TestSuite` class are valid here too. So, if we run our new suite with the following command:

```
java net.scriptinginjava.ch7.MyGroovySuite
```

we should expect a result similar to this one:

```
..F..F
Time: 0.15
There were 2 failures:
1) testIteration(net.scriptinginjava.ch7.MyTest)
junit.framework.AssertionFailedError:
Element greater than 5 found!
  at net.scriptinginjava.ch7.MyTest.testIteration(
    MyTest.java:28
  )
  at sun.reflect.NativeMethodAccessorImpl.invoke0(
    Native Method
  )
  at sun.reflect.NativeMethodAccessorImpl.invoke(
```

```
    NativeMethodAccessorImpl.java:39
  )
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(
    DelegatingMethodAccessorImpl.java:25
  )
  at chapter7.MyGroovySuite.main(MyGroovySuite.java:21)
2) testIteration(net.scriptinginjava.ch7.MyGroovyTest)
junit.framework.AssertionFailedError:
Element greater than 5 found!
  at gjdk.net.scriptinginjava.ch7.MyGroovyTest_GroovyReflector.invoke(
    MyGroovyTest_GroovyReflector.java
  )


FAILURES!!!
Tests run: 4,  Failures: 2,  Errors: 0
```

Because we have two identical test cases defined, there are two failures (one in each case).

Scripts as Unit Test Cases

Besides integration with JUnit, you can use Groovy to write tests as regular scripts. Groovy defines the `assert` command that you can use for this purpose. This method is demonstrated in the following code snippet (`testScript.groovy`):

```
testList = [1,3,5,7]
assert testList.size() == 4
```

As you can see, this simple script could serve as a unit test. The only limitation is that there is only one signature of this method (which was not the case with `assert` methods described earlier), but for simple use cases, this is more than enough.

To demonstrate what happens in a case of unexpected value, let's modify the original script as follows:

```
testList = [1,3,5,7]
assert testList.size() == 3
```

If you now run it with:

```
groovy testScript.groovy
```

you get the following error on the screen:

```
Caught: java.lang.AssertionError: Expression:
  (testList.size() == 3)
    at testScript.run(testScript.groovy:2)
    at testScript.main(testScript.groovy)
```

Summary

As you saw in this section, using scripting for conducting unit tests provides greater flexibility and enables you to write tests faster. However, these tests take more time to execute. Whether your project benefits from this technique primarily depends on the development process you're using.

Interactive Debugging

As mentioned, unit testing has changed the traditional process of software development, as developers now write test cases first and implement the functionality afterward. Even though most of the functionality in our applications is covered by appropriate test cases, there are situations where interactive shells could still be helpful.

In the process of writing your classes, regardless of whether you have written tests for them, you often need to check whether a certain code snippet (or subfunctionality) behaves properly, or you need to do some fine-tuning. Sometimes it is too time consuming to run tests just for these purposes, and sometimes it is just not possible (or not worthwhile) to do it.

In such situations, Java programmers are forced to make their classes executable (by defining the `main()` method in them), perform some initial testing (debugging), and then delete that code after they are sure that they can proceed.

Using a `main()` method for initial testing and debugging is neither natural nor efficient. Also, this kind of debugging usually lasts longer than you would like and could result in forgotten `main()` methods that should not exist in your application.

A much more appropriate way to perform initial testing and debugging is by using interactive shells and scripting languages. While you are implementing a certain functionality, you can use the interactive shell (available for every scripting language covered in the book thus far) for a quick test of your class's behavior for some test values. In that way, you can work around certain problems and doubts that you have in your design.

Note

You should not consider this as a replacement for unit testing, and it should not interfere with that process by any means. It is just an additional tool for developers that should enable them to create more-robust solutions faster.

Let's demonstrate this on a simple example. Imagine that we have to write a `User` class with a `setEmail()` method. This method should throw an `Exception` if the submitted email address is not in a valid format. There are numerous ways to verify that an email address is valid, but let's say that we want to use a regular expression for this task.

First, not many people can write a nontrivial regular expression correctly in their first attempt. Even if you can find an off-the-shelf solution on the Internet, test it (and adapt it to your needs) before making it an integral part of your application.

So, before we start coding our business logic class, we want to write a regular expression that we can use. Even if most of you are thinking of unit tests right now, a much better solution is available for you to use for this task. Unit tests are important for checking basic functionalities and boundary value behaviors, and to provide a mechanism for ensuring that this functionality does not break in a year or two. But it is not convenient to use unit tests as a tool for writing, checking, and fine-tuning regular expressions.

Imagine, for example, that we have come up with the following regular expression as a solution for email address verification:

```
^[a-zA-Z][\\w\\.-]*[a-zA-Z0-9]@[a-zA-Z0-9][\\w\\.-]
*[a-zA-Z0-9]\\.[a-zA-Z][a-zA-Z\\.]*[a-zA-Z]$
```

Many people create a dummy class or a dummy `main()` method in an existing class just as a helper tool for playing with the regular expression. For example, let's see what happens if we use a local e-mail address (without a domain):

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

    if (Pattern.matches(
       "^[a-zA-Z][\\w\\.-]*[a-zA-Z0-9]@"
     + "[a-zA-Z0-9][\\w\\.-]"
     + "*[a-zA-Z0-9]\\.[a-zA-Z][a-zA-Z\\.]*[a-zA-Z]$"
     , "dejan")) {
        System.out.println("true");
    } else {
        System.out.println("false");
    }

}
```

As you can see, this simple method just prints `true` or `false` indicating whether the test value matches the pattern. If the class with this `main` method is run in this case, it prints the following:

```
false
```

This means that this regular expression does not accept users on the local host. In our example, we assume that this is desirable behavior. If it wasn't, we would be forced to dig into the expression and try to fix the problem. Then we would have to run the application repeatedly until it worked. We would repeat this process for all the cases that are of interest to us.

As mentioned, however, this approach is not flexible enough, and we should not use it for these purposes. Instead, let's now try to use Groovy's interactive shell to perform initial testing of the regular expression. We can just run the interactive shell by using the `groovysh` command and start playing (see Listing 7.11).

Listing 7.11 Interactive Debugging with the Groovy Shell

```
$ groovysh
Let's get Groovy!
================
Version: 1.0-beta-8 JVM: 1.5.0_01-b08
Type 'exit' to terminate the shell
Type 'help' for command help
Type 'go' to execute the statements

groovy> p = "^[a-zA-Z][\\w\\.-]*[a-zA-Z0-9]@[a-zA-Z0-9][\\w\\.-]
*[a-zA-Z0-9]\\.[a-zA-Z][a-zA-Z\\.]*[a-zA-Z]$"
groovy> println "dejan" ==~ p
```

Listing 7.11 Continued

```
groovy> go
false

groovy>
```

It's easy to try out other test values. Let's try the valid e-mail address, for example:

```
groovy> println "dejan@nighttale.net" ==~ p
groovy> go
true
```

As we can see, the regular expression seems to work fine in this case. We can now play further, by passing an address with a wrong domain name:

```
groovy> println "dejan@nighttale" ==~ p
groovy> go
false
```

This also fails. We can proceed like this until we are sure that this regular expression is the right solution for us.

An interactive shell in combination with Groovy's support for Perl-like syntax regular expressions helps us to write and test regular expressions much faster. There is no need to recompile the class and run it for every little change in the expression. The Groovy shell keeps a history of recently used statements, so you can easily return to the expression and make all your desired changes.

Now that we have an initial version of the regular expression, we can write our `User` class as follows:

```
package net.scriptinginjava.ch7;

import java.util.regex.Matcher;
import java.util.regex.Pattern;


public class User {

    Pattern emailPattern = Pattern.compile(
        "^[a-zA-Z][\\w\\.-]*[a-zA-Z0-9]@[a-zA-Z0-9][\\w\\.-]
      + "*[a-zA-Z0-9]\\.[a-zA-Z][a-zA-Z\\.]*[a-zA-Z]$");
```

```
    String email;

    public void setEmail(String email) throws Exception {
        Matcher matcher = emailPattern.matcher(email);
        if (!matcher.matches()) {
            throw new Exception(
                "Not valid email address: " + email
            );
        }
        this.email = email;
    }

}
```

Of course, we still have to write the appropriate unit tests for this method and cover all the necessary cases of its use (some of the techniques presented in the "Unit Testing" section earlier in the chapter could be used for this task). The interactive shell just helped us to get going, create an expression that compiles, and do the desired job at first glance.

Naturally, we could write test cases for the `setEmail()` method first and use them to build the complex regular expression. But it would take much more time to get the initial behavior used in the application. Also, we can use the shell to prove some concept before we even can be sure that it could be used in our project. Only when we have settled that the concept is correct can we start planning a testing strategy and implementation details.

Another situation where BeanShell and Groovy could be particularly helpful is the design of complex algorithms and code snippets. Again, it is usually not convenient to do such things in the targeted class. Instead, you could move such snippets to the script (since the language syntax is pretty much the same), and you could work to prove your concepts there more easily. When they are good enough, you can return them to the Java class and expose them to unit testing.

The final use of interactive shells discussed in this section is exploration of new APIs. In the Java community today, there are many open source projects, and you often need to evaluate them to see whether they match your requirements. In other cases, you just want to get a feeling for the API, such as which classes are there and how they are used. Again, interactive

shells (and similar tools) give you all the freedom you need to play with new libraries while you are reading the documentation. Very quickly, you can put up some basic examples and get a feeling for the API. After this first learning phase, continue to think of how you can integrate the API into your project.

In this section, I covered only a few simple examples of the use of interactive shells in the development process. You might feel that these issues are not that important in these modern days of extreme programming, but the next time you find yourself in a situation similar to one I described, try using an interactive shell. I think it will become one of your new habits.

Build Tools (Ant Scripting)

In Chapter 2, “Appropriate Applications for Scripting Languages,” we concluded that a build tool such as Make is essential for successful project deployment. We also said that Ant, due to its advantages, is the number one build tool for Java projects.

> **NOTE**
> If you don’t already have it, you can obtain Ant from the Apache software foundation site (http://ant.apache.org/). There you can also find installation instructions for various platforms.

Before we go into more detail on Ant, let’s do a quick overview. Tasks, such as `<javac>`, which compiles Java source files, are implemented as Java classes. The developer, or any other person responsible for project deployment, creates an XML file (usually called `build.xml`) in which he composes those tasks and adapts them to the desired environment. In the following code, you can find a typical example of these XML definition files:

```
<project name=”MyProject” default=”deploy” basedir=”.”>

    <property name=”src” value=”src” />
    <property name=”build” value=”classes” />
    <property name=”jar.name” value=”my.jar” />
    <path id=”lib”>
        <pathelement path=”${basedir}/”/>
        <fileset dir=”${basedir}/lib”/>
    </path>
    <pathconvert property=”cpath” targetos=”unix”
     refid=”lib” />
    <pathconvert property=”manifest.cpath”
        refid=”lib” pathsep=” “
    />
    <property name=”build.compiler” value=”modern”/>
```

```
    <property name="main.class" value="Test"/>

    <target name="clean">
        <delete dir="${build}" />
    </target>

    <target name="prepare">
        <mkdir dir="${build}" />
    </target>

    <target name="compile" depends="prepare">
        <javac srcdir="${src}" destdir="${build}"
            classpath="${cpath}" deprecation="on"
        />
        <copy todir="${build}">
            <fileset dir="${src}">
                <exclude name="**/*.java"/>
            </fileset>
        </copy>
    </target>

    <target name="deploy" depends="compile">
         <jar jarfile="lib/${jar.name}">
             <fileset dir="${build}"/>
             <manifest>
                 <attribute name="Main-Class"
                     value="${main.class}"/>
                 <attribute name="Class-Path"
                     value="${manifest.cpath}" />
             </manifest>
         </jar>
         <delete dir="${build}" />
    </target>
</project>
```

Here is how this XML file is structured:

- The `<project>` tag is the root tag of these build files. Its attributes can define the project name, the location of the project on the filesystem, and the default target (more on this in a second).
- With the `<property>` and similar tags, you can define the parameters of your project, such as the location of the source files within it.
- `<target>` tags are used to define the actual actions that we want to take in the build process. In this example, we have targets such as "compile the project" (target `compile`) or "create a JAR distribution of the project" (target `deploy`, which is a default target for this project).

When you run Ant, it looks for the `build.xml` file in the current folder (or one specified with the `-buildfile` switch). If no target is specified, the default one is executed.

So if you type:

```
ant
```

you can expect output similar to the following:

```
Buildfile: build.xml

prepare:

compile:
    [javac] Compiling 1 source file to
C:\eclipse\workspace\Book\chapter7\classes

deploy:
      [jar] Building jar:
C:\eclipse\workspace\Book\chapter7\lib\my.jar
   [delete] Deleting directory
C:\eclipse\workspace\Book\chapter7\classes

BUILD SUCCESSFUL
Total time: 4 seconds
```

This approach is usually good enough for simple projects, but it is also the task where a scripting language would be a good solution. Some people feel constrained by XML and need more flexibility; others just don't like to "program" using XML's angle bracket syntax. Even James Duncan Davidson, the creator of Ant, wrote (http://x180.net/Articles/Java/AntAndXML.html):

In retrospect, and many years later, XML probably wasn't as good a choice [for writing Ant build files] as it seemed at the time. I have now seen build files that are hundreds, and even thousands, of lines long and, at those sizes, it turns out that XML isn't quite as friendly a format to edit as I had hoped for. As well, when you mix XML and the interesting reflection-based internals of Ant that provide easy extensibility with your own tasks, you end up with an environment which gives you quite a bit of the power and flexibility of a scripting language—but with a whole lot of headache in trying to express that flexibility with angle brackets.

...

If I knew then what I know now, I would have tried using a real scripting language, such as JavaScript via the Rhino component or Python via JPython, with bindings to Java objects which implemented the functionality expressed in today's tasks. Then, there would be a first class way to express logic and we wouldn't be stuck with XML as a format that is too bulky for the way that people really want to use the tool.

You can use scripting languages in two different places to make Ant more flexible. The first is the programming logic. Even though you can use conditional logic in Ant, it is cumbersome to write conditions (or any other nontrivial logic) using XML syntax. Consider the following example:

```
<if>
    <!– "if" evaluates this element –>
    <bool>
        <and>
            <available file="build.xml"/>
            <available file="run.xml"/>
        </and>
    </bool>

    <!– if true, then tasks listed here will execute –>
    <echo>build.xml and run.xml are available</echo>

    <!–
      if false, then tasks inside the "else" will execute
    –>
    <else>
        <echo>
          didn't find one or both of build.xml and
          run.xml
        </echo>
    </else>
</if>
```

This example shows how even a simple `if-else` conditional structure could be cumbersome to write in XML. So if you ever feel the need for looping, recursive processing of some data, and other programming elements in your build process, scripting languages could help you. Of course, you could always write an Ant extension and try to use it for that purpose, but as we saw in the preceding example, such a solution is not natural or easy to use. Because it's easy to embed scripting languages

into XML documents and scripting languages naturally support programming structures, you have much more flexibility if you use a scripting language.

Another way you can use scripting languages with Ant is for extending its functionality. Without scripting languages, the only way to extend Ant's build process is through custom tasks. These tasks, as we said, are implemented as Java classes. This approach is not flexible enough to support all situations.

In this section, we cover two approaches for integrating Ant and scripting languages:

- Support for the BSF that enables you to write and evaluate scriptlets in build files
- Groovy's `AntBuilder` markup class that you can use to define build files with GroovyMarkup syntax

BSF Support

In Chapter 6, "Bean Scripting Framework," we saw how to integrate the BSF and technologies such as JSP and XSLT. You use a similar principle to integrate the BSF library with Ant. You can embed script expressions into XML build files and evaluate them when you're building your project.

To use the BSF, you have to provide the appropriate JARs for Ant to use. Those JARs are the appropriate version of `bsf.jar` and an implementation of the scripting engine you are going to use. Prior to version 1.6, Ant used IBM's version of the BSF. Apache's version is used from BSF version 1.6 and on. Refer back to Chapter 6 for more information on how to obtain these versions of the BSF.

These external libraries can be supplied in one of two ways:

- They can be placed in the `$ANT_HOME/lib` folder where they are picked up by Ant automatically.
- They can be made available through the system `CLASSPATH` environment variable.

We are going to use the JavaScript language in the examples in this section. Therefore, the appropriate `bsf.jar` and

js.jar files should be placed in the $ANT_HOME/lib folder. Again, if you are using BSF version 2.4 or later, you also have to put commons-logging.jar in this folder too.

You define scripting elements in the build files within the <script> tag (see Listing 7.12).

Listing 7.12 Ant BSF Support Example

```
<project name="MyProject" default="hello">

    <target name="hello">
        <script language="javascript">
            <![CDATA[
                importPackage(java.lang, java.util, java.io);
                System.out.println("Hello World");
            ]]>
        </script>
    </target>
</project>
```

In Listing 7.12, we defined a simple build file that consists of only one default target, named hello. Inside this target, we defined our script. The language that we are going to use is defined with the language attribute of the <script> tag, just as it was with the BSF integrations we did earlier. Actual statements of the script are defined inside the following element:

```
<![CDATA[ script definition ]]>
```

You can also use the src attribute to specify the location of the external script as a file:

```
<script language="javascript" src="script.js">
```

In our simple example, we imported some Java libraries and printed the welcome message on the screen.

If you run this build file by typing:

```
ant
```

on the command line, you should get the following result:

```
Buildfile: build.xml

hello:
```

```
   [script] Hello World

BUILD SUCCESSFUL
Total time: 0 seconds
```

Even though you can probably find useful applications of general scripting support in build files, the context of the Ant project is what can give you the real power.

To understand this better, let's examine Listing 7.13.

Listing 7.13 Advanced Ant BSF Support Example

```
<project name="MyProject" default="hello">

<property name="srcdir" value="src"/>

    <target name="hello">
        <script language="javascript">
            <![CDATA[
                importPackage(java.lang, java.util, java.io);
                importPackage(Packages.org.apache.tools.ant);
                importPackage(
                    Packages.org.apache.tools.ant.taskdefs
                );

                System.out.println("srcdir="
                    + self.getProject().getProperty("srcdir"));
                System.out.println("This project has the "
                    + "following targets defined:");
                tar = MyProject.targets.keys();
                while (tar.hasMoreElements()) {
                    System.out.println(tar.nextElement());
                }
                call.execute();
            ]]>
        </script>
    </target>

    <target name="call">
        <echo message="Call"/>
    </target>
</project>
```

First, we need to import the appropriate Ant packages (`org.apache.tools.ant` and `org.apache.tools.ant.taskdefs`) to use Ant's API in the script. Now let's see which variables are available to us.

The `self` variable represents an instance of the actual `<script>` task. This task is implemented with the `org.apache.tools.ant.taskdefs.optional.Script` class, which extends the standard Ant's `org.apache.tools.ant.Task`.

NOTE

We do not cover Ant's API in more detail here. You cannot find Ant's API documentation on the Apache Web site, but it is included with all the distributions under the `$ANT_HOME/docs/manual/api` folder.

You can use the `getProject()` method to obtain a reference to the current project (`org.apache.tools.ant.Project`), and you can use the `getProperty()` method of the `Project` class to get the value of the property defined in the file. In this example, we printed the value of the `srcdir` property defined earlier in the file.

Besides the `self` variable, you can use variables named after the project and targets defined in it. Thus, we can use the `MyProject`, `hello`, and `call` variables in the script. The `MyProject` variable, named after the project, is equivalent to the one obtained with the `self.getProject()` call. The `hello` and `call` variables are instances of the `org.apache.tools.ant.Target` class. We can execute another target defined in the file simply by calling the `execute()` method of the appropriate variable. In this example, we executed the `call` target at the end of the script.

If we run Ant with this build file, we should expect the following result to show up:

```
Buildfile: build.xml

hello:
   [script] srcdir=src
   [script] This project has the following targets defined:
   [script] hello
   [script] call

     [echo] Call


BUILD SUCCESSFUL
Total time: 0 seconds
```

BSF support for Ant makes build files more "programmable." Also, you can extend the build files via inline scripts (without having to write a separate Java class for every required extension).

GroovyMarkup (AntBuilder)

BSF support gives us more flexibility in writing build files. But scripting languages are still used for writing just small fractions of such files. Those who would like to avoid XML in general

and use a real scripting language can find a solution in Groovy's `AntBuilder` class.

In Chapter 5, "Advanced Groovy Programming," we saw how to use GroovyMarkup syntax to create and process treelike structures with closures and named parameters. Because Ant build files, like every other XML-like structure, are classic tree-like structures, they are suitable for this type of processing.

In Listing 7.14, you can find an alternative representation of the build file example defined at the beginning of this section.

Listing 7.14 AntBuilder Example

```
import org.apache.tools.ant.Task

class SimpleBuild {

    def ant = new AntBuilder()

    def name = "MyProject"
    def defaultTask = "deploy"
    def basedir = "C:\\eclipse\\workspace\\Book\\chapter7"
    def src = basedir + "\\src"
    def build = basedir + "\\classes"
    def jarName = "my.jar"
    def mainClass = "net.nighttale.SampleClass"
    def classpath = ant.path(id:"lib") {
        pathelement(path:"${basedir}/")
        fileset(dir:"${basedir}/lib")
    }

    def clean() {
        ant.delete(dir:"${build}")
    }

    def prepare() {
        ant.mkdir(dir:"${build}")
    }

    def compile() {
        prepare()
        ant.javac(
            srcdir:"${src}", destdir:"${build}",
            classpath:"${classpath}",deprecation:"on"
        )
        ant.copy(todir:"${build}") {
            fileset (dir:"${src}") {
                exclude(name:"**/*.java")
            }
        }
    }

    def deploy() {
        compile()
         ant.jar(jarfile:"lib/${jarName}") {
```

Listing 7.14 Continued

```
            fileset(dir:"${build}")
            manifest {
                attribute(name:"Main-Class", value:"${mainClass}")
            }
        }
        ant.delete(dir:"${build}")
    }

    static void main(args) {
        def b = new SimpleBuild()
        b.run(args)
    }


    void run(args) {
        if ( args.size() > 0 ) {
            defaultTask = args[0]
        }
        invokeMethod(defaultTask, null)
    }
}
```

As you can see, no XML is used here. We have defined a Groovy class with an `ant` property (which is an instance of the `AntBuilder` class). This property allows us to execute Ant tasks in the Groovy fashion. Parameters are passed as named parameters, so the following call:

```
ant.delete(dir:"${build}")
```

is equivalent to the following task:

```
<delete dir="${build}" />
```

Subtasks are defined within the closure of the current task. Thus, the following call:

```
ant.copy(todir:"${build}") {
   fileset (dir:"${src}") {
      exclude(name:"**/*.java")
   }
}
```

is equivalent to this task definition:

```
<copy todir="${build}">
       <fileset dir="${src}">
```

```
            <exclude name="**/*.java"/>
      </fileset>
</copy>
```

We have defined targets as methods of this class, and Ant's properties are now properties of the class.

All that is left to do now is to write the `main()` method that calls the appropriate target (method of the class), and we have a fully functional build script.

If you run this script with the following command:

```
groovy SimpleBuild.groovy
```

you can expect the following output (which is similar to the output produced by the equivalent XML build file from the beginning of this section):

```
 [mkdir] Created dir:
   /office/work/int/users/dejanb/temp/classes
 [javac] Compiling 1 source file to
   /office/work/int/users/dejanb/temp/classes
 [copy] Copying 1 file to
   /office/work/int/users/dejanb/temp/classes
 [jar] Building jar:
   /home/office/work/int/users/dejanb/temp/lib/my.jar
 [delete] Deleting directory
   /office/work/int/users/dejanb/temp/classes
```

The real power of this approach is in the fact that you can use any Groovy statement in build files like this. Calls to Ant tasks are just one Groovy feature you can use.

Let's demonstrate this with a simple example. If you have ever worked on a Web application development project in Java, you know how convenient it is to be able to manage your Web server (servlet container) during the deployment process.

Tomcat (http://jakarta.apache.org/tomcat/), Apache's implementation of the servlet container, provides Ant tasks that you can use to install, start, and stop your Web application. You do this through the manager application that you can access via HTTP. All you need is the URL, username, and password for the manager application, and the path to your Web application. Ant tasks form HTTP requests to the application and process returned results.

The typical build file (written using `AntBuilder`) that uses these tasks could look like that shown in Listing 7.15.

Listing 7.15 Advanced AntBuilder Example

```
import org.apache.catalina.ant.*
import org.apache.tools.ant.Task

class Build {
    def ant = new AntBuilder()

    def url = "http://localhost:8080/manager"
    def username = "admin"
    def password = "tomcat"
    def path = "/testapp"

    def Build() {
        ant.taskdef(name: "list"
            , classname: "org.apache.catalina.ant.ListTask")
        ant.taskdef(name: "start"
            , classname: "org.apache.catalina.ant.StartTask")
        ant.taskdef(name: "stop"
            , classname: "org.apache.catalina.ant.StopTask")
    }

    static void main(args) {
        build = new Build()
        build.all()
    }

    def start() {
        ant.start(url: "${url}", username: "${username}"
            , password: "${password}", path: "${path}")
    }

    def stop() {
        ant.stop(url: "${url}", username: "${username}"
            , password: "${password}", path: "${path}")
    }

    def deploy() {
        // build and deploy
    }

    def all() {
        stop()
        deploy()
        start()
        }

    def list() {
        ant.list(url: "${url}", username: "${username}"
            , password: "${password}")
    }

}
```

First, we have to define the tasks we are going to use. These tasks are implemented with classes located in the `org.apache.catalina.ant` package. After that, we are free to create targets (methods) that are simply going to call these tasks.

You might have noticed the `all()` method that stops the application, rebuilds it, and then starts it again. Basically, this is just a higher-level task that groups more tasks in a common routine. But because this is the operation most frequently used in the Web application development process, it is practical and useful for a developer. Thus, it is set to be our default task in this build file.

The preceding `all()` method works fine, until your application is not running for some reason. Usually this occurs because your previous deployment failed due to an error. In that case, the `stop()` task fails with the following error:

```
[stop] OK - Stopped application at context path /testapp
[start] OK - Started application at context path /testapp

[stop] FAIL - Encountered exception
  java.lang.IllegalStateException: standardHost.stop
  /testapp:
  LifecycleException:  Container StandardContext[/testapp]
  has not been started
```

Because of this, the default method is not working in the desired way, and we have to call the `deploy` and `start` tasks manually. For this reason, we cannot use this task to start a Web application from scratch (or after an unsuccessful deployment). It would be much more convenient if we could use this default task regardless of whether the application is running. Unfortunately, we can do nothing to customize this behavior if we are using XML build files.

But because Groovy provides a real programming environment, we can deal with this issue in a more flexible manner. For example, we can catch this error, handle it, and proceed with the rest of the tasks as normal.

The following code snippet contains the modified `all()` method that catches errors that can occur while the application is being stopped:

```
def all() {
    try {
        stop()
    } catch (BuildException ise) {
        println "Application has not been started"
    }
    deploy()
    start()
}
```

If we now run the `all` task when the application is not running, we get output similar to that shown in the following code:

```
[stop] FAIL - Encountered exception
  java.lang.IllegalStateException: standardHost.stop
  /testapp:
  LifecycleException:  Container StandardContext[/testapp]
  has not been started
  Application has not been started
[start] OK - Started application at context path /testapp
```

> **Note**
> We cannot perform this kind of error handling with Ant itself, and this is just a simple example of the new possibilities that scripted build files can bring to you.

As you can see, we caught the exception, and because this is not a fatal error, we proceeded further with application deployment.

A similar approach is built into the Rake (http://rake.rubyforge.org/) tool. Rake is a build tool similar to the `make` program that we briefly described in Chapter 2. The only difference is that Rake uses the Ruby scripting language to declare tasks and dependencies, which provides much more flexibility to the build process.

Summary

In this section, we covered two approaches that you can use to make the project-building process more dynamic. With one approach, we can define and execute scriptable elements inside the XML definition file. The other approach avoids XML completely and uses the Groovy programming language as a build tool.

Knowing these methods can be useful in situations when you find yourself limited by the original Ant's functionality. Which approach you use, and how you use it, depends, of course, on your requirements and preferences.

Shell Scripting

A *shell* is a sort of interface to the UNIX operating system. Although it is primarily used as a command interpreter, it is considered a programming language as well.

In Chapter 2, we saw how we could compose an application of filter programs using pipes. To make those programs reusable, we need a mechanism to create a script and make it executable. For these purposes, the UNIX shell allows the creation of so-called *interpreter files*.

> **NOTE**
> In case you are using the Windows platform, you can set a UNIX-like environment with Cygwin (www.cygwin.com/). It allows you to run examples described in this section.

An interpreter file begins with a line that has the following form:

```
#!path_to_interpreter [argument]
```

`path_to_interpreter` represents an interpreter used to process the rest of the file. Note that only one argument can be optionally passed to the interpreter. In the following code, you can see an example of the shell script that executes a Java application:

```
#!/bin/bash

java -cp /home/dejanb/application.jar Main
```

Here, the `/bin/bash` shell interpreter is used to process the file. This simple script just executes the `java` command (with the provided classpath and class that contains the application).

Suppose that we named this script file `run`. To make it executable, we must have permission to execute it:

```
$ chmod +x run
```

Now we can run it by typing the following in the command line:

```
$ ./run
```

We can use the same principle to make our script files (written in Groovy, BeanShell, or any other scripting language) executable on the UNIX platform. Suppose, for example, that Groovy is installed in the /opt/groovy folder. You can create a UNIX executable script file in Groovy like this:

```
#!/opt/groovy/bin/groovy

println "Hello world"
```

Now make it executable (the script filename is assumed to be run):

```
chmod +x run
```

You can now run it like an ordinary UNIX shell script:

```
./run
```

Classpath

If you use the preceding approach, you might encounter a problem, however: You cannot change the classpath for the script in the first line. Remember the syntax that is used to specify the interpreter; it accepts only one argument, and there is no way to specify the switch for the interpreter.

The script uses the global classpath set by the CLASSPATH environment variable. With BeanShell, you can deal with this problem in another way. BeanShell scripts are loaded by a special class loader, which provides the addClassPath() and setClassPath() commands that can be used to modify the classpath directly from the script. For example, the following line includes the specified JAR file in the classpath, and we are free to use classes defined in it further on:

```
addClassPath("/home/dejanb/application.jar");
```

With this approach, your scripts' users don't have to configure the classpath manually; you can do that work for them.

Example

Making UNIX executable scripts with languages that can interact with existing Java modules could be useful for many applications. In this section, we discuss one scenario where you can simplify your application using the BeanShell.

Many developers tend to think that only one programming language is more than enough for all types of problems. In many development areas, however, that approach is simply not possible. One obvious example is a standalone Java application that should be used in the UNIX environment.

In this example, we create a simple application stub, just to make sure that all the necessary application elements are present. The application just starts an XML-RPC server and makes a connection to the database. It is a typical configuration of the Java server application. The XML-RPC - XML Remote Procedure Call (www.xmlrpc.org) protocol is just one of the ways to expose your application's API to other applications via the Internet. We're using it here just as an example. A Java application with a similar architecture could use REST, SOAP, or any other of the available mechanisms for the same purpose. Even servlet containers (such as Jetty or Tomcat) could represent this kind of application, and you could apply a similar architecture, problem, and solution to them too.

Let's see what sort of Java application handles this task. The first important element is the configuration file. Two of the most common solutions for the configuration files are the property files and the XML configuration files. Here, we are going to use the standard Java property file format that holds a simple mapping between the property and its value in the text file. In the following listing, you can see how this configuration file looks for our application (`myApp.properties`):

> **NOTE**
> I've omitted the actual logic of the application because it is not important for this topic.

```
jdbcDriver = com.mysql.jdbc.Driver
jdbcUrl = jdbc:mysql://localhost/groovy
jdbcUsername = root
jdbcPassword =

xmlRpcPort = 8888
```

This file contains a configuration for a JDBC database connection, and the port number on which the XML-RPC server listens.

Next, we need a class that loads this configuration file and initialize necessary objects. This is the main class of the application, the one started through its `main()` method.

```
import java.io.FileInputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Properties;

import org.apache.xmlrpc.WebServer;

public class MyApp {

    public static void main(String[] args) throws Exception {
        //load properties
        Properties props = new Properties();
        FileInputStream propFile =
            new FileInputStream("myApp.properties");
        props.load(propFile);
        // initialize XML-RPC server
        int port =
            new Integer(
              props.getProperty("xmlRpcPort")
            ).intValue();
        WebServer server = new WebServer(port);
        //add handlers here
        server.start();
        //initialize JDBC connection (or pool)
        Class.forName(props.getProperty("jdbcDriver"));
        Connection con =
            DriverManager.getConnection(props.getProperty("jdbcUrl"),
                props.getProperty("jdbcUsername"),
                props.getProperty("jdbcPassword"));

        //create business logic objects

    }
}
```

Now we need to provide an easy way to start this application. In an environment where users are not familiar with Java, it is hard to expect them to set the classpath properly, and it is even harder to expect them to start it directly with the `java` command:

```
java -cp "classes/:lib/xmlrpc-1.2-b1.jar:\
lib/mysql-connector-java-3.1.4-beta-bin.jar" MyApp
```

A common solution is to create a shell script that sets the classpath and run the application:

```
#!/bin/bash

java -cp “classes/:lib/xmlrpc-1.2-b1.jar:\
lib/mysql-connector-java-3.1.4-beta-bin.jar” MyApp
```

As you can see, even if you want to use only Java to create your application, you can end up using scripting for some of its parts.

Now look at the BeanShell script in Listing 7.16.

Listing 7.16 Shell Scripting Example

```
#!/opt/java/bin/java bsh.Interpreter
addClassPath(“lib/mysql-connector-java-3.1.4-beta-bin.jar”);
addClassPath(“lib/xmlrpc-1.2-b1.jar”);
import java.io.FileInputStream;
import java.sql.Connection;
import java.sql.DriverManager;

import org.apache.xmlrpc.WebServer;

//configuration
jdbcDriver = “com.mysql.jdbc.Driver”;
jdbcUrl = “jdbc:mysql://localhost/groovy”;
jdbcUsername = “root”;
jdbcPassword = “”;

xmlRpcPort = 8888;

//application
WebServer server = new WebServer(xmlRpcPort);
server.start();
//initialize JDBC connection (or pool)
Class.forName(jdbcDriver);
Connection con = DriverManager.getConnection(jdbcUrl
     , jdbcUsername, jdbcPassword);
//create business logic objects
```

This simple script replaces all three of the elements in the typical Java server application that would run on a UNIX-like operating system. First, we made this script executable and set the classpath according to our needs. Next, we have the configuration section, which is practically a replacement of the properties file used in the preceding example. Finally, we have the application initialization logic that uses variables set in the configuration section and initializes components of the application.

As you have probably noticed, this solution is both simpler and easier to write and maintain than the previous one.

Administration and Management

Every system, regardless of whether it is a UNIX server or a Java Web application, needs to be maintained. For UNIX servers, system administrators found scripting languages were an ideal tool for automating tasks that needed to be executed. In Chapter 2, we discussed use of schedulers together with scripting languages to accomplish these tasks successfully.

In this section, we implement the solution to the administration problem set in Chapter 2. To summarize:

- We are developing a highly trafficked Web site in Java.
- Data on the users of the Web site is stored in the database, in the `login` table, with the following structure:

```
create table login (
    userid int NOT NULL auto_increment,
    username varchar(32) not null,
    password varchar(32) not null,
    last_login date not null,
    status varchar(16) not null default 'ACTIVE',

    primary key(userid)
);
```

  As you can see, besides the regular types of fields usually stored in this kind of table, this table contains the `last_login` field, which indicates the last date when the user logged on to the system.

- After a while, you get the request from your customer (a Web site owner) who wants to periodically delete users who haven't logged in for a month.

The same question from Chapter 2 stands. What shall we use to solve this problem: an EJB component or a five-line script?

Let's try to use the second approach and see what benefits it can bring. First, we need to create a script to perform this task. I decided to use Groovy and its GroovySQL module, so here is my script (see Listing 7.17).

Listing 7.17 Administration Script Example

```
import groovy.sql.Sql
import java.util.Calendar

if (args.length != 0)
    days = args[0]

sql = Sql.newInstance("jdbc:mysql://localhost/webapp"
     , "com.mysql.jdbc.Driver")

Calendar now = Calendar.getInstance()

now.add(Calendar.DAY_OF_MONTH, -new Integer(days))

sql.executeUpdate("UPDATE login SET status = 'DELETED' \
                    WHERE status ='ACTIVE' AND last_login < ?"
                  , [now.time] )

println "${sql.updateCount} user(s) have been marked as deleted"
```

You could execute this script on its own or from some Java application. In the first case, the days variable is set from an argument passed to the script. If that script is embedded in the application, this variable is bounded to the script.

Now we can proceed, and run the script manually from time to time and delete users that match the criteria:

```
groovy /opt/scripts/users_delete.groovy 30
```

The script should print the number of deleted users on the screen:

```
3 user(s) have been marked as deleted
```

This was simple, and as you can see, it does not interfere with the application in any way. The changes in the script do not require the application to be redeployed, and because this is a simple task, there is no need for robust transaction management or any of the other tools that are needed for building a robust Web application.

As mentioned in Chapter 2, no administration task is fun to perform more than twice, so the next natural step is to try to automate this task. For that, we need to use an appropriate scheduler, and because we are building a Java solution, we are

going to use a scheduler we can integrate with it. Many Java schedulers are available today, but unfortunately, I didn't find any that support script execution scheduling. So this is a good opportunity to see how we can customize a scheduler to support scripting. You can use a principle similar to this one for any other Java application or library.

In this example, we are going to use the Quartz scheduler (www.opensymphony.com/quartz/), which is a solid solution. We are not going to discuss it in more detail than is necessary to understand what we are trying to accomplish here. For more details about Quartz, consult its documentation.

Quartz provides the `org.quartz.Job` interface that is used to make Java components executable by the scheduler. This simple interface is an implementation of the command pattern:

```
package org.quartz;

public interface Job {

    public void execute(JobExecutionContext context)
            throws JobExecutionException;

}
```

To create BSF support for Quartz, we need to create an instance of the `BSFManager` class and implement the `execute()` method, which evaluates the desired script. As you can see from the definition of the interface, parameters to the job are passed using the `org.quartz.JobExecutionContext` class. Because we want to execute the script file, we force the `filename` parameter to be passed to the job required.

In Listing 7.18, you can find an implementation of a Quartz job that is capable of executing script files (in any language supported by the BSF).

Listing 7.18 The Script Job Example

```
package net.scriptinginjava.ch7;

import java.io.FileReader;
import java.util.Iterator;

import org.apache.bsf.BSFManager;
import org.apache.bsf.util.IOUtils;
import org.quartz.JobDataMap;
```

Listing 7.18 Continued

```
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;
import org.quartz.Job;

public class ScriptJob implements Job {

    BSFManager manager = new BSFManager();

    public ScriptJob() {
        BSFManager.registerScriptingEngine(
                "groovy",
                "org.codehaus.groovy.bsf.GroovyEngine"
                , new String[] { "groovy", "gy" }
            );
    }

    public void execute(JobExecutionContext context)
        throws JobExecutionException {
        JobDataMap dataMap = context.getJobDetail()
            .getJobDataMap();
        String fileName = dataMap.getString("filename");
        if (fileName == null)
            throw new JobExecutionException(
                "Script file must be defined"
            );

        try {
            Iterator it =
                context.getJobDetail().getJobDataMap().keySet()
                    .iterator();
            while (it.hasNext()) {
                Object key = it.next();
                Object value = dataMap.get(key);
                manager.declareBean(
                    (String) key, value, value.getClass());
            }

            String language =
                BSFManager.getLangFromFilename(fileName);
            String script =
                IOUtils.getStringFromReader(
                    new FileReader(fileName)
                );
            manager.exec(language, fileName, 0, 0, script);
        } catch (Exception e) {
            throw new JobExecutionException(e);
        }
    }

}
```

In the constructor of the `ScriptJob` class, we registered the Groovy programming language. We did this just to be sure that the language is registered in case someone wants to use this class with BSF version 2.2 or older.

In the `execute()` method, we first check whether the `filename` attribute has been set. Next, we iterate through all the attributes that are set for the job, and we bind them to the scripting engine. Finally, we load the script from the file and evaluate it.

Now that we have a scheduler capable of evaluating script files, we can configure it to run our `users_delete.groovy` script. Quartz enables us to define the job configuration in the XML file. The XML file (`jobs.xml`) in Listing 7.19 instructs the scheduler to execute the `/opt/scripts/users_delete.groovy` script file every first of the month, at midnight. The value for the `days` variable passed to the script is set to 30, which means it deletes all users that are inactive for more than 30 days.

Listing 7.19 Quartz Scheduler Configuration Example

```
<?xml version='1.0' encoding='utf-8'?>

<quartz xmlns="http://www.quartzscheduler.org/ns/quartz"

       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

       xsi:schemaLocation="
http://www.quartzscheduler.org/ns/quartz
http://www.quartzscheduler.org/ns/quartz/
job_scheduling_data_1_1.xsd"

       version="1.1">


  <job>
    <job-detail>
      <name>deleteUsers</name>
      <group>adminJobs</group>
      <job-class>net.scriptinginjava.ch7.ScriptJob</job-class>

      <job-data-map allows-transient-data="true">

            <entry>
                 <key>filename</key>
                 <value>/opt/scripts/users_delete.groovy</value>
            </entry>

            <entry>
                 <key>days</key>
                 <value>30</value>
            </entry>

        </job-data-map>
    </job-detail>
    <trigger>
      <cron>
        <name>deleteUsers</name>
```

Listing 7.19 Continued

```
        <group>adminJobs</group>
        <job-name>deleteUsers</job-name>
        <job-group>adminJobs</job-group>
        <cron-expression>0 0 1 * * *</cron-expression>
      </cron>
    </trigger>
  </job>

</quartz>
```

Now it's time to start the scheduler. In our real Web application, we would do this through the specialized servlet located in the `org.quartz.ee.servlet.QuartzInitializerServlet` class. Here, we create a simple class that initializes and starts a scheduler:

```
package net.scriptinginjava.ch7;

import org.quartz.Scheduler;
import org.quartz.SchedulerFactory;

public class Start {

    public static void main(String[] args) throws Exception {
        SchedulerFactory schedFact =
            new org.quartz.impl.StdSchedulerFactory();
        Scheduler sched = schedFact.getScheduler();
        sched.start();
    }

}
```

In this example, we created a new instance of the scheduler factory and obtained the default scheduler from it.

Regardless of whether you are going to use it in a standalone or a Web environment, the Quartz scheduler needs a `quartz.properties` file for proper configuration:

```
# Configure Main Scheduler Properties

org.quartz.scheduler.instanceName = TestScheduler
org.quartz.scheduler.instanceId = AUTO

# Configure ThreadPool

org.quartz.threadPool.class =
org.quartz.simpl.SimpleThreadPool
org.quartz.threadPool.threadCount = 3
```

```
org.quartz.threadPool.threadPriority = 5

# Configure JobStore

org.quartz.jobStore.misfireThreshold = 60000

org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore

# Configure Plugins

org.quartz.plugin.jobInitializer.class =
org.quartz.plugins.xml.JobInitializationPlugin
org.quartz.plugin.jobInitializer.fileName = jobs.xml
org.quartz.plugin.jobInitializer.overWriteExistingJobs =
true
org.quartz.plugin.jobInitializer.failOnFileNotFound = true
org.quartz.plugin.jobInitializer.validating = false
org.quartz.plugin.jobInitializer.validatingSchema = true
org.quartz.plugin.jobInitializer.scanInterval = 10
```

Among other settings, here you can instruct the scheduler to use the `jobs.xml` file from earlier. The file is scanned for changes every 10 seconds, and its configuration is updated accordingly.

Here we have created a simple environment in which we can easily schedule scripts to be executed in precisely specified time intervals. An environment like this could be valuable for quick creation of the various kinds of reports and administration tasks needed in Java applications (similar to the one I described here).

Conclusion

In Chapter 2, we covered the use of scripting languages for various tasks in the development and maintenance of the information infrastructure. In this chapter, we saw how we could apply that knowledge in Java-related projects.

Chapter 8 focuses on the successful use of scripting in the applications architecture. We cover some of the design patterns that employ scripting languages to make software projects more flexible and easier to build and maintain.

SCRIPTING PATTERNS

Knowledge of a particular programming language (or programming platform) is usually not enough to build a successful software project. Also required is a solid understanding of the project architecture. Before coding a project with a graphical user interface, every developer must ask himself the same questions—among them, how he is going to build the user interface, and how he is going to refresh it when application data changes. Knowing the answers to such project architecture questions up front enables software architects and developers to solve the recurring architectural problems they encounter with every new project.

Over time, developers began collecting these recurring software design challenges and documenting their solutions. This description of a recurring software design problem along with its solution is called a *design pattern*.

As design patterns began to increase in complexity, developers introduced a unique format for presenting them. This unique format is useful because it encourages a well-defined structure that emphasizes all the relevant aspects of the pattern and helps developers to understand it in a minimum of time. Today, every design pattern description consists of the following four basic elements:

- **Name**—Helps to create a commonly understood design vocabulary by using known terms to describe recurring problems and their well-understood solutions.
- **Problem**—Describes a particular recurring problem that a pattern tries to solve. It also defines a context in which a problem exists.
- **Solution**—Describes elements and their relationships that lead to efficient design, which solves the problem.
- **Consequences**—Describe the impacts of applying the pattern in the system. They contain results and trade-offs introduced by the solution.

We follow this format to document the patterns presented in this chapter because it is widely accepted by the programming community.

Software design patterns became popular in the 1990s with the book *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley Professional Computing Series). Written by a group of authors known as the “Gang of Four” (GoF), the book focuses on object-oriented programming design and is a good introduction to object-oriented design and design patterns. Over time, many authors created patterns for specific development areas (such as J2EE platform patterns), with the same goal of documenting solutions for recurring problems in software architecture.

This chapter does not cover the basic set of design patterns; for more information on that topic, consult the aforementioned *Design Patterns* book. Instead, we extend some of the well-known design patterns and see how the script-programming paradigm fits in. We also introduce a few new patterns specific to systems that are fully or partially developed using scripting languages.

I start by explaining the existing patterns and providing an example of each in Java. After this basic introduction, I discuss elements that can employ scripts and look at the benefits and drawbacks of the scripting approach. We use the Groovy programming language to code scripted parts of our patterns.

Table 8.1 lists the patterns we cover in this chapter, along with a short description of the problems they solve.

Table 8.1 Scripting Patterns

Pattern	Description
Scripted Components	How to compose an application out of reusable components (written in a system-programming language) with scripts
Mediator (Glue Code)	How to create flexible many-to-one relationships among components
Script Object Factory	How to improve the runtime performance of scripting solutions in the production environment
Observer (Broadcasters)	How to create flexible one-to-many relationships among components
Extension Point	How to extend components' behavior with simple-to-write scripts
Active File	How to use scripting to store both data and the code that handles that data in the file

Scripted Components Pattern

This pattern explains how to compose an application out of reusable components (written in a system-programming language) with scripts.

Problem

Component-oriented software design introduces reusable components that are context free and that rely as little as possible on other components in the system. They are designed to *play* a specific role in systems and to be independent of the application context. As such, developers can reuse them among different projects.

The term *component* can mean different things, depending on the context in which it is used. Sometimes authors treat a single class implemented in a system-programming language as the component. In other cases, this term refers to a whole subsystem with a unified interface (API) to the rest of the system.

Enterprise JavaBeans (EJBs), which were introduced in the J2EE specification, are another example of software components. Every EJB component consists of a few interfaces to the container and classes that implement their business logic. Such components are then deployed to the application container (server), which manages them and is responsible for their life cycle.

It doesn't matter whether you are going to work in an environment that will force you to compose applications out of well-defined components, or whether you are going to follow those practices yourself. In either case, the component-oriented architecture separates the development process into the following two phases:

- Designing and implementing the components
- Creating an application by composing the components

The goal of the Scripted Components design pattern is to easily compose applications from components. In many organizations, senior developers are responsible for component design tasks, and inexperienced (junior) programmers usually assemble applications out of those components. In those situations, we need to provide mechanisms for easy manipulation of software components, which leads to flexible systems and rapid development.

Software architects could also define another goal for their projects. They may want to be able to rearrange and adapt components at runtime. This ability generally introduces greater flexibility of the whole architecture and makes further changes less painful.

Solution

In the first two chapters of this book, we compared scripting and system-programming languages. In that discussion, I said that system-programming languages are a good tool for implementing system components. This statement is true, mostly because system-programming languages offer good runtime performance and a well-defined structure that forces developers to strictly implement defined interfaces.

Although system-programming languages are good for implementing system components, scripting languages are an excellent solution for implementing the glue code, whose main responsibility is to wire components together and to mediate in their communication. Developers use scripting languages for this task for several reasons. One of the reasons developers cite most often is that scripting languages have an easy-to-learn

syntax that enables inexperienced programmers to assemble applications quickly out of well-defined existing components. Gluing the components also requires less code in scripting, thus making it easy for other developers to understand and maintain the application.

Another reason for their wide adoption is that script development skips the compilation phase. This means you can modify the component wiring without having to rebuild the whole project (you can even modify it during runtime).

Because this book focuses on the Java platform, we can treat Java libraries (APIs) as software components for the purposes of our discussion. One of the major advantages of the Java platform is actually the existence of these libraries (most of them created by the open source community). The wide range of projects available to Java developers for almost any problem domain makes application development much easier. Usually all you have to do is just pick the desired APIs (and frameworks) and glue them together.

That is where the scripting languages covered in this book come into play. You can use them to flexibly compose applications out of preexisting components. With their dynamic typing, powerful data structures, and other characteristics, they are a much better choice than Java for component composition and initialization.

Also, if the scripting paradigm is adopted by a development organization, senior developers could be involved in the development, testing, and improvement of components (both open source and those that are built in-house). Junior developers, meanwhile, could be involved in implementing the client's requirements. Senior developers also could benefit from scripting, meaning that they can finish component composition tasks faster.

Consequences

This pattern has the following consequences:

- It encourages a component-oriented system architecture.

- It introduces greater flexibility in adapting and arranging system components.
- It introduces an additional performance overhead in the application because the gluing part of the application will be interpreted, which is certainly slower than executing compiled code with equivalent functionality.

Sample Code

As an example of this design pattern, we can use the sample code discussed back in Chapter 7, "Practical Scripting in Java," when we were talking about UNIX scripting with BeanShell (see Listing 8.1).

Listing 8.1 Scripted Component Pattern Example

```
#!/opt/java/bin/java bsh.Interpreter
addClassPath("lib/mysql-connector-java-3.1.4-beta-bin.jar");
addClassPath("lib/xmlrpc-1.2-b1.jar");
import java.io.FileInputStream;
import java.sql.Connection;
import java.sql.DriverManager;

import org.apache.xmlrpc.WebServer;

//configuration
jdbcDriver = "com.mysql.jdbc.Driver";
jdbcUrl = "jdbc:mysql://localhost/groovy";
jdbcUsername = "root";
jdbcPassword = "";

xmlRpcPort = 8888;

//application
WebServer server = new WebServer(xmlRpcPort);
server.start();
//initialize JDBC connection (or pool)
Class.forName(jdbcDriver);
Connection con = DriverManager.getConnection(
     jdbcUrl, jdbcUsername, jdbcPassword
);
//create business logic objects
```

As discussed in Chapter 7, we can think of the XML-RPC library and the JDBC driver as components. These components encapsulate some of the fundamental logic of our application. Beside components, we need code that will arrange components to gain the appropriate application behavior. We often refer to this code as an application because we look at the components

as black boxes with the specified functionality. In Listing 8.1, an application is practically a BeanShell script that configures, initializes, and composes these components.

The fact that we wrote this script to be executable on UNIX platforms is not crucial for this pattern. We could have used any scripting language (that can use Java classes) and run the script using its interpreter. The point here is that we arranged the compiled components using the scripting language of our choice.

Related Patterns

The Scripted Components pattern encourages us to initialize and configure our components using scripting languages. If you need these components to be interconnected, see the sections on the Mediator and Observer patterns.

Mediator Pattern (Glue Code Pattern)

This pattern explains how to create flexible many-to-one relationships among components.

Problem

The Mediator pattern discussed in this section is an extension of the original Mediator pattern, one of the patterns from the original set discussed in the GoF book. This modified pattern is often referred to as the Glue Code pattern in the scripting community. The original Mediator pattern encourages a component-oriented architecture that is flexible and easily adapted to different applications. It helps developers to centralize complex relationships between objects and thus simplify modification of system behavior.

To understand the benefits of the Mediator pattern, it helps to put the problem it solves into context. A complex system contains many components and, usually, many more interconnections among them. The first idea that comes to a developer's mind is to make every component (object) responsible for its connections toward other components in the system. In such a design approach, it is likely that every component will end up

in relation with all the other components that it is using. As a result, the reusability of that component will be lost because it is tightly coupled to the context of the application in which it is used. Furthermore, an effort to change the system behavior could be tedious because it is distributed among many components (in other words, all over the application).

A common example of this problem in action concerns the construction of complex graphical user interfaces. A complex user interface consists of many widgets, including buttons, list boxes, text fields, and so on, which are often heavily interconnected. Actions performed on one widget usually lead to a changed state (a change in values) on another widget.

For example, say we have to create a dialog box that consists of a list box, text field, and button. The dialog box has to satisfy requirements that are usually found in these kinds of UI widgets: When the list box item is selected, it should be displayed in the text field, and when certain items are selected, the button must be disabled.

Without an appropriate mediator, a list box must have a reference to these two components. Now let's say that we want to add another widget to the dialog. If both the list box and the button must interact with the new widget, we have to add another widget reference to these components. Also, their logic will change to support actions that involve this newly added widget.

This example illustrates that every change in the user interface requires a change in many components. Such a design leads to tightly coupled components and hard-to-change systems that behave much like a monolithic system. Another problem is that the behavior of the dialog box is distributed among its components, making it more likely that we will forget to change some of them. Thus, there is a higher probability of introducing new bugs into the system.

Solution

The original Mediator pattern solves this problem by encapsulating the collective behavior of a group of components into a separate object (component). All other components reference only the mediator that is responsible for all the interactions in the group of objects.

To begin, let's discuss the original Mediator pattern and then see how scripting can be used in its context. To demonstrate this pattern's elements, and how they collaborate, we use a dialog box example similar to the one I just described. The dialog will be used for choosing a desired font. It consists of a list box, with all the currently available fonts in the system, and a label that displays sample text in a selected font.

The Mediator pattern suggests that we keep a list box and the label components separate, and that we create an additional class that will encapsulate the actual dialog logic. Previously, we had a list box that was responsible for setting the font of the label component directly. Now, it will notify the mediator object that its value changed, and the mediator will react to that event by changing the font of the text displayed in the label.

Figure 8.1 shows an interaction diagram for all components included in the font dialog. In this diagram, you can see that

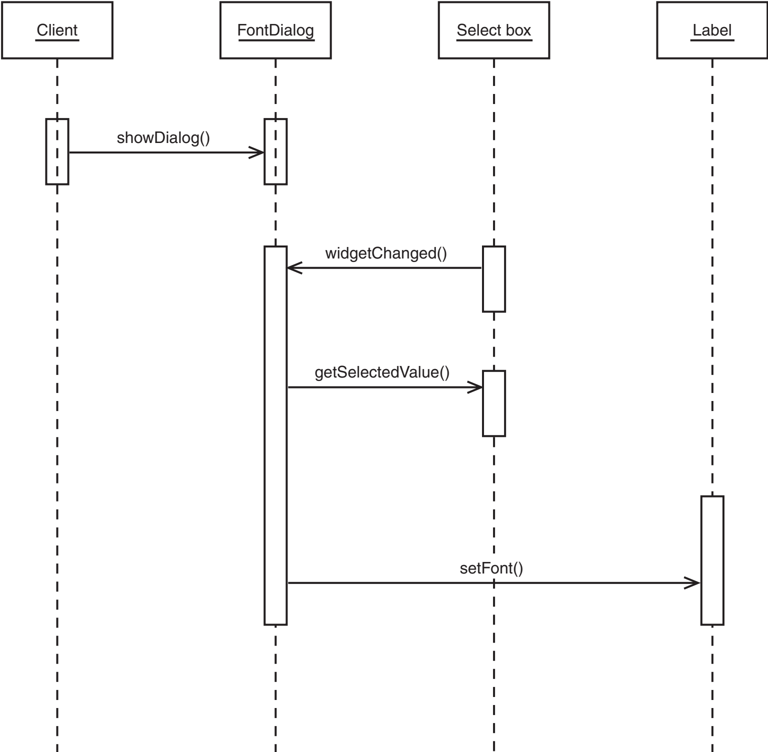


FIGURE 8.1 Mediator pattern example—interaction diagram

the `FontDialog` component is a mediator between a list box and a label. The list box notifies the font dialog of a change, and it exposes a method that the mediator will use to identify the change and get the new value (the font name in this case). Then the font dialog takes an appropriate action on the other component by setting a chosen font.

Because we centralized behavior of the dialog box, it is much easier now to add another component to the dialog. The list box and label component remain untouched, and we have only to adjust the font dialog component (the mediator).

In a programming environment that uses scripting languages, you can modify the original Mediator pattern to achieve greater flexibility. In some literature that covers design patterns for scripting languages, authors refer to this extended Mediator pattern as the Glue Code pattern.

On top of the original Mediator pattern, the Glue Code pattern suggests that you use scripts to configure the components' behavior. Again, as in the original pattern, the components are initialized with the appropriate mediator, and that is the only component that they are aware of. In the Glue Code pattern, the mediator is a script that is executed when the action occurs.

Traditionally, the Glue Code pattern was implemented by putting into the component a reference to the script interpreter and the script location. The component would then evaluate the script or call a function defined in one. Which action it took usually depended on the particular language and environment being used. As in the original Mediator pattern, the component had to pass itself to the mediator so that it could identify what future action to take. If the mediator was a plain script, the component would bind itself to the scripting engine. If the mediator behavior was encapsulated in a function, the component would pass itself as an argument.

Nowadays, however, many scripting languages that can be integrated into the Java platform are capable of implementing Java interfaces with scripts. This means the mediator can be defined by the interface, and we can provide its implementation in the script.

With this in mind, the Glue Code pattern differs from the original Mediator pattern only in the way in which the mediator

class is loaded. In the Mediator pattern, the client instantiates the mediator directly, using its constructor. In "script-aware" Glue Code pattern, the client loads the class using the language interpreter, as we saw in previous chapters.

In the sample code that demonstrates this pattern, which is provided later in this section, we force the second approach (implementation of a mediator as a Java interface in Groovy), because it is a more natural solution for the Java environment.

Consequences

The original Mediator pattern has the following consequences.

- It localizes behavior that otherwise would be distributed among several objects.
- It emphasizes the reusability of components because they are decoupled and all application-specific logic is located in the mediator.
- It simplifies object interaction because it replaces many-to-many relationships with one-to-many interactions.
- It centralizes control in such a way that the complexity of the component interaction is traded for the complexity of the mediator.

All these consequences apply to the Glue Code pattern too. Additionally, the Glue Code pattern introduces the following consequences.

- The behavior of the component and its interaction with other components is not fixed and can be changed during runtime.
- Because low-level components are wired using a script, this pattern introduces additional runtime overhead.

Sample Code

We begin by demonstrating an implementation of the Mediator pattern in Java. Then, we modify that example to adapt it to the scripting environment. In that way, we can see the differences between the Mediator and Glue Code approaches and be better able to determine how to apply them in different contexts.

The most common example of the Mediator pattern (even in the GoF book) is the implementation of a complex dialog box. In this section, we create the simple dialog box described earlier. Note that Java's Swing and AWT packages implement other patterns that we could use for this task, but here we will force the Mediator pattern for demonstration purposes.

First, we create an interface for our mediator component. This interface will be implemented with the Java class in the Mediator pattern approach. In the Glue Code solution that follows, we will implement it with the Groovy script (see Listing 8.2).

Listing 8.2 Mediator Interface

```
package net.scriptinginjava.ch8.mediator;

public interface DialogDirector {

    public void widgetChanged(Widget widget);

    public void showDialog();

}
```

The interface contains two methods:

- The `showDialog()` method is used by the client, and its purpose is to initialize and show the dialog box.
- The `widgetChanged()` method is crucial. It is called every time the widget changes its value, and as you can see, the widget is passed as an argument to this method. The `widgetChange()` method contains the complete logic of the dialog box.

Because our widgets have to be aware of the mediator, we will create an abstract `Widget` class that will be a wrapper around the standard Swing components (see Listing 8.3).

Listing 8.3 Mediator-Aware Widget Abstraction

```
package net.scriptinginjava.ch8.mediator;

import javax.swing.JComponent;

public abstract class Widget {

    DialogDirector director;
```

Listing 8.3 Continued

```
    public Widget(DialogDirector director) {
        this.director = director;
    }

    public void changed() {
        director.widgetChanged(this);
    }

    public abstract JComponent getComponent();

}
```

This class contains an instance of the `DialogDirector` interface implementation, which is passed as an argument to the constructor. The `changed()` method calls the `widgetChanged()` method of the mediator, with itself as an argument. Every component that extends this class is responsible for calling this method when changing its state. The abstract `getComponent()` method is a helper method that the mediator can use to access Swing components.

Now we can implement the label and list box components that we will use in the dialog box (and throughout the application), as shown in Listing 8.4 and Listing 8.5.

Listing 8.4 Label Widget

```
package net.scriptinginjava.ch8.mediator;

import javax.swing.JComponent;
import javax.swing.JLabel;

public class Label extends Widget {

    private JLabel label;

    public Label(DialogDirector director, String text) {
        super(director);
        label = new JLabel(text);
    }

    public JComponent getComponent() {
        return this.label;
    }
}
```

The `Label` component is simple because it does not actively participate in the interaction. We have just wrapped the `javax.swing.JLabel` class in our `Widget` abstract class defined earlier (see Listing 8.3).

Listing 8.5 List Box Widget

```
package net.scriptinginjava.ch8.mediator;

import java.awt.Font;
import java.awt.GraphicsEnvironment;

import javax.swing.JComponent;
import javax.swing.JList;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;

public class FontListBox extends Widget
     implements ListSelectionListener {

     private JList fonts = new JList();

     public FontListBox(DialogDirector director) {
          super(director);
          Font[] allFonts =
               GraphicsEnvironment
                    .getLocalGraphicsEnvironment()
                         .getAllFonts();
          String[] fontNames = new String[allFonts.length];
          for (int i=0;  i< allFonts.length; i++) {
               fontNames[i] = allFonts[i].getName();
          }
          fonts.setListData(fontNames);
          fonts.addListSelectionListener(this);
     }

     public JComponent getComponent() {
          return fonts;
     }

     public void valueChanged(ListSelectionEvent e) {
          changed();
     }
}
```

The `Label` component is more complicated because it has to notify the mediator that the user selected a certain item. Besides the `Widget` class, it also implements the `javax.swing.event.ListSelectionListener` interface. This interface is registered with the actual `JList` component as its list selection listener. The `valueChanged()` method (defined in this interface) is called every time a user selects a certain item. This method simply calls the `changed()` method defined in the `Widget` superclass, and further notifies a director (through the `widgetChanged()` method) that the event has occurred.

Now it is time to implement the mediator. It, of course, implements the `DialogDirector` interface. This class also

extends the `javax.swing.JFrame` class because it represents a dialog box that holds other components (see Listing 8.6).

Listing 8.6 Mediator Implementation

```
package net.scriptinginjava.ch8.mediator;

import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.Font;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JScrollPane;

public class FontDialog extends JFrame
    implements DialogDirector {

    Label sample;
    FontListBox fontList;
    JScrollPane listScroller;

    public FontDialog() {
        setTitle("Font dialog");
        getContentPane().setLayout( new FlowLayout() );
        setSize( 400, 200 );


        fontList = new FontListBox(this);
        listScroller = new JScrollPane(fontList.getComponent());
        listScroller.setPreferredSize(new Dimension(250, 80));
        getContentPane().add(listScroller);

        sample = new Label(this, "Sample text");
        getContentPane().add(sample.getComponent());
    }

    public void widgetChanged(Widget widget) {

        if (widget == fontList) {
            String fontName =
                (String)(
                    (JList)widget.getComponent()
                ).getSelectedValue();
            Font font = new Font(fontName, Font.PLAIN, 12);
            JLabel sampleText = (JLabel)sample.getComponent();
            sampleText.setFont(font);
        }

    }

    public void showDialog() {
        show();
    }

}
```

In the constructor of the `FontDialog` class, we initialized the label and list components. Note that both components are initialized with the instance of the `FontDialog` class. The `showDialog()` method just calls the `show()` method of the `JFrame` parent class, which makes it visible on the screen.

As mentioned earlier, all the code used to glue components together is located in the `widgetChanged()` method. In this simple example, the method checks whether the component in the argument is the `fontlist` component defined in the dialog. If it is, the mediator gets the selected font name and changes the font of the `sample` label component. With this approach, the list box and label components are not aware of each other, and coordination of their communication is done through the `widgetChanged()` method of the `DialogDirector` interface.

At the end, we implement the client that will use this dialog box (see Listing 8.7).

Listing 8.7 Mediator Client

```
package net.scriptinginjava.ch8.mediator;

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

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

public class App implements ActionListener {


    public void actionPerformed(ActionEvent e) {
        FontDialog fontDialog = new FontDialog();
        fontDialog.showDialog();
    }

    public static void main(String[] args) {

        JFrame frame = new JFrame("Application");
        frame.getContentPane().setLayout( new FlowLayout() );
        frame.setSize( 100, 75 );

        JButton button = new JButton("Select font");
        button.addActionListener(new App());
        frame.add(button);
        frame.show();
    }
}
```

This simple client shows the frame with the button (see Figure 8.2).

When the button is clicked, the font dialog box appears (see Figure 8.3).

You can try to select a certain font from the list and see how it affects the text in the label (see Figure 8.4).

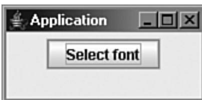


FIGURE 8.2 Application frame

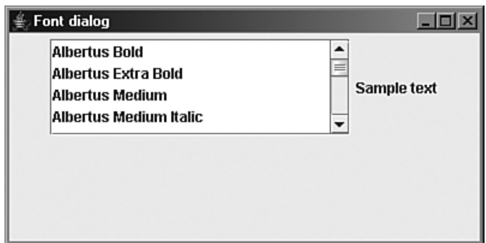


FIGURE 8.3 Font dialog

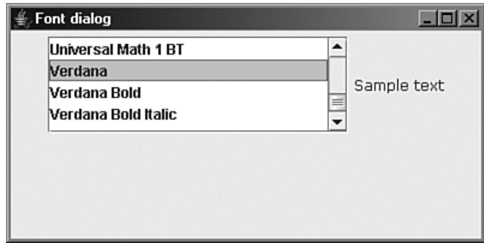


FIGURE 8.4 Font selected in the dialog

Note that the font dialog is instanced directly as a regular class through its constructor.

Now it is time to introduce scripting to this example. Because Groovy's syntax is close to Java's (refer to Chapter 4, "Groovy," for more details), we can rename `FontDialog.java` to `FontDialogScript.groovy` (because there are no incompatibilities) and thus convert the Mediator pattern to the Glue Code pattern.

Of course, first we must change how the font dialog in the client is initialized to support this Groovy implementation (see Listing 8.8).

Listing 8.8 Glue Code Client

```
package net.scriptinginjava.ch8.mediator;

import groovy.lang.GroovyClassLoader;

import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import java.lang.reflect.Constructor;

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

public class ScriptApp implements ActionListener {

    public void actionPerformed(ActionEvent e) {
       GroovyClassLoader loader = new GroovyClassLoader();
        try {
            Class mediatorClass = loader.parseClass(
                new File(
                    “net/scriptinginjava/ch8/mediator/”
                  + “FontDialogScript.groovy”
                )
            );
            Constructor mediatorConstructor =
                mediatorClass.getConstructor(new Class[]{});
            DialogDirector fontDialog =
                (DialogDirector) mediatorConstructor.newInstance(
                    new Object[] {}
                );
            fontDialog.showDialog();
        } catch (Exception ex) {
            ex.printStackTrace();
        }

    }

    public static void main(String[] args) {

        JFrame frame = new JFrame(“Application”);
        frame.getContentPane().setLayout( new FlowLayout() );
        frame.setSize( 100, 75 );

        JButton button = new JButton(“Select font”);
        button.addActionListener(new ScriptApp());
        frame.add(button);
        frame.show();
    }
}
```

We used a `groovy.lang.GroovyClassLoader` class to parse the script and initialize a Java object from it (we discussed this technique in Chapter 4). Other than that, the whole application remains absolutely the same as before. If we start it, we will get the same client (with the same behavior) as that shown in Figure 8.2.

The big difference is that now we can change the behavior of the mediator at runtime. For example, you can modify the script in the following fashion (while the application is running), as shown in Listing 8.9.

Listing 8.9 Modified Mediator Component

```
package net.scriptinginjava.ch8.mediator;

import java.awt.Dimension;
import java.awt.GridLayout;
import java.awt.Font;

import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JScrollPane;

public class FontDialogScript extends JFrame
    implements DialogDirector {

    Label sample;
    FontListBox fontList;
    JScrollPane listScroller;

    public FontDialogScript() {
        setTitle("Font dialog");
        getContentPane().setLayout( new GridLayout() );
        setSize( 400, 200 );


        fontList = new FontListBox(this);
        listScroller = new JScrollPane(fontList.getComponent());
        listScroller.setPreferredSize(new Dimension(250, 80));
        getContentPane().add(listScroller);

        sample = new Label(this, "Sample text");
        getContentPane().add(sample.getComponent());
    }

    public void widgetChanged(Widget widget) {

        if (widget == fontList) {
            String fontName =
                (String)(
                    (JList)widget.getComponent()
                ).getSelectedValue();
```

Listing 8.9 Continued

```
            Font font = new Font(fontName, Font.PLAIN, 12);
            JLabel sampleText = (JLabel)sample.getComponent();
            sampleText.setFont(font);
        }

    }

    public void showDialog() {
        show();
    }

}
```

If you now click the button that shows a dialog, you get a somewhat modified dialog box (see Figure 8.5).

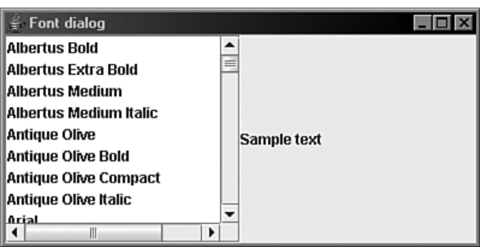


FIGURE 8.5 Modified Font dialog

This approach obviously could be useful in the development process because we can model the behavior of the mediator component (that contains the complete application segment logic) without having to recompile or even restart the application.

Related Patterns

As noted earlier, because low-level components are wired using the script, this pattern introduces an additional runtime overhead. You can solve this problem using the Script Object Factory pattern, described in the following section.

Script Object Factory Pattern

This pattern explains how to improve the runtime performance of scripting solutions in the production environment.

Problem

As discussed in Chapters 4 and 5, you can use some scripting languages to implement Java interfaces. But although that is a valuable technique during development, it incurs a performance overhead because the interpreted implementation is slower than the compiled one in most cases. We can avoid that overhead if we compile scripts to Java classes when producing our system.

Refer back to the example that we implemented for the Mediator pattern. We used Groovy to implement the mediator component in a more flexible way. That technique enabled us to modify the behavior of our component while the application was running, and thus increased our development speed. The price we pay for that flexibility, however, is an additional performance overhead.

After we have our component modeled out, we want to avoid this performance overhead, and so we might want to use the component's compiled version. The Script Object Factory pattern helps us to develop systems in which we can easily switch between scripted and compiled component implementations.

Solution

The Script Object Factory pattern is an extension of the original GoF Factory Method pattern. The Factory Method pattern encourages us not to instantiate objects directly but to use a method of a factory object for this job. With this approach, deciding which concrete object will be created depends on the subclass of the factory object. Also, the type of object that will be created might depend on the value of some application parameter.

The Factory Method pattern will help us to achieve our goal. We will use an application parameter (which we can obtain from the configuration file, for example) to decide whether an object will be loaded from the script or as a compiled class.

So during development, this method will be instructed to evaluate scripts, and our application will be easily changeable. However, we will set the Factory method to instantiate objects

directly from the compiled scripts during system deployment. In this way, we can achieve both the flexibility of development with a scripting language and the performance of a pure Java solution.

Consequences

The original Factory Method pattern has the following consequences:

- The client is not bound to the application-specific classes. With the Factory Method, the client deals only with the appropriate interface.
- Object creation with this method is more flexible than it is through direct instantiation.

The Script Object Factory pattern additionally introduces the following:

- It introduces all the benefits of rapid application development with scripting languages, such as runtime modification of component behavior.
- It provides the component's system-programming language performance in the production environment.
- It hides the actual details of object creation.

Sample Code

To demonstrate this technique, we use the font dialog manager that we used when discussing the Mediator pattern. All we have to do is to modify our client class. We make it able to use both scripted and compiled versions of the mediator component. The parameter given in the property file (`app.properties` in this application) defines which version it should use.

To start, we have to be sure that a compiled script (a class) is available for the application. For that purpose, we have to modify the Ant target responsible for compiling the sources of our application (the `compile` target in this example; see Listing 8.10).

Listing 8.10 Ant Task That Compiles All Scripts Inside the Project

```
<project name="groovy project" default="compile">
    <taskdef
        name="groovyc"
        classname="org.codehaus.groovy.ant.Groovyc"
    />

    <target name="compile">
        <javac srcdir="." destdir="/dev/project/classes"/>
            <groovyc
                srcdir="."
                destdir="/dev/project/classes"
                classpath="/dev/project/classes"
                listfiles="true"
            />
    </target>
</project>
```

As you can see, we added a definition of the `groovyc` task and used it, along with the `javac` task, to compile all the Groovy source files it finds in the source directory (and its subdirectories). In this way, we are sure that other classes in our application can reach the compiled script.

Now we need a mechanism to determine whether to use the scripted or compiled version of the mediator. As I said, we use the property value to configure this application's behavior. In this case, the property file is simple and contains just one value:

```
debug = true
```

If the value of the `debug` property is `true`, the client will try to load the class by evaluating the script file. Otherwise, it will use a precompiled class, which was created by compiling the script in the project's building process (see Listing 8.11).

Listing 8.11 Scripted Object Factory Pattern Example

```
package net.scriptinginjava.ch8.mediator;

import groovy.lang.GroovyClassLoader;

import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import java.io.FileInputStream;
import java.lang.reflect.Constructor;
import java.util.Properties;

import javax.swing.JButton;
```

Listing 8.11 Continued

```
import javax.swing.JFrame;

public class ScriptApp implements ActionListener {

    private Properties props = new Properties();

    public ScriptApp() throws Exception {
        FileInputStream propFile = new FileInputStream(
            "app.properties"
        );
        props.load(propFile);
    }

    public void actionPerformed(ActionEvent e) {
        try {
            DialogDirector fontDialog = getFontDialog();
            fontDialog.showDialog();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    public static void main(String[] args) throws Exception {

        JFrame frame = new JFrame("Application");
        frame.getContentPane().setLayout( new FlowLayout() );
        frame.setSize( 100, 75 );

        JButton button = new JButton("Select font");
        button.addActionListener(new ScriptApp());
        frame.add(button);
        frame.show();
    }

    public DialogDirector getFontDialog() throws Exception {
        String debug = props.getProperty("debug");
        if (Boolean.parseBoolean(debug)) {
            GroovyClassLoader loader = new GroovyClassLoader();
                Class clazz = loader.parseClass(
                    new File(
                        "net/scriptinginjava/ch8/"
                      + "mediator/FontDialogScript.groovy"
                    )
                );
                Constructor constr = clazz.getConstructor(
                    new Class[]{}
                );
                DialogDirector fontDialog =
                    (DialogDirector)constr.newInstance(
                        new Object[] {}
                );
                System.out.println("script");
                return fontDialog;
        } else {
            System.out.println("byte");
            return new FontDialogScript();
        }

    }
}
```

In its constructor, our client loads properties from the `app.properties` file. If you set the value of the `debug` property to `true`, you can modify the behavior of the `FontDialogScript` mediator class at runtime. You can do this by changing the appropriate script, just as we did in the Mediator pattern example. After we have an acceptable behavior (or the application is about to be shipped to the client), there is no need to pay this performance overhead anymore. We can change the value of the `debug` property (to `false`), recompile the project, maybe pack it in a JAR file, and deploy it. It now performs like a standard Java application, and the customer should not notice any performance differences.

Related Patterns

The benefit that we gathered with this pattern is additional flexibility in the development phase of our application, but without the performance penalties paid in the production of such a solution. As such, it could be used with practically every scripting pattern that encourages implementation of Java interfaces with scripts. In this section, we have seen how we can use it together with the Mediator pattern. I will leave it to you as an exercise to try modifying the Observer pattern example, which we see in the following section, to use the Script Object Factory pattern as well.

Observer (Broadcasters) Pattern

This pattern explains how to create flexible one-to-many relationships among components.

Problem

Earlier in this chapter, I explained how you could adapt the Mediator pattern to the scripting environment. In this section, we do the same with the original GoF Observer pattern. Some authors who document patterns for scripting languages call this extension the Broadcasters pattern.

With the Mediator pattern, we solved the problem of many-to-one interconnections among system components. In many situations, you will find it necessary to enable a component to notify other system components when a certain event triggers. The intent of the original Observer pattern is to define mechanisms that we can use to create these one-to-many kinds of relationships among components (objects).

An example of the Observer pattern can be found, again, in the development of complex graphical user interfaces. Many development toolkits (frameworks) that exist today encourage developers to separate application data from their presentation in the user interface. This separation introduces many benefits in terms of system development. First, it allows us to apply different views to the same data easily. We can use the same data objects to initialize different views, and thus we can create new views quickly. Also, without data objects being tightly coupled to their views (and application logic in general), it becomes much easier to reuse these data objects among different projects.

To explain how the Observer pattern works, I use an imaginary business application. This business application has a dialog box in it that shows the price and the tax value of an item in stock. That dialog box must have two label fields, one for the total price and another for the tax part of the price. If the price of the item changes, both fields must be updated. Our task is to provide a flexible mechanism for that operation.

Solution

To achieve this separation, the Observer pattern defines two kinds of components:

- The *subject* component can have many observers. In our introductory example, the `Price` object could be seen as the subject.
- *Observer* components are interested in receiving information when the subject changes its state. In the preceding example, the dialog labels could be modeled as the observers.

To explain the object structure in the Observer pattern in more detail, we discuss its implementation in Java.

In the `java.util` package, you can find the `Observable` class and the `Observer` interface, which define abstractions necessary for implementation of this pattern. The `Observable` class provides mechanisms for registration and notification of the registered `Observers`. The notification takes place when the `Observable` object (the subject) changes its state. This class has to be extended by all subjects, such as the `Price` class in the introductory example.

To enable registration to the subject (the `Observable` object), all observers (such as labels in the price example) must implement the `Observer` interface. This interface defines only one method that is used for notification purposes, as we will see in a moment. The structure diagram of these classes is shown in Figure 8.6.

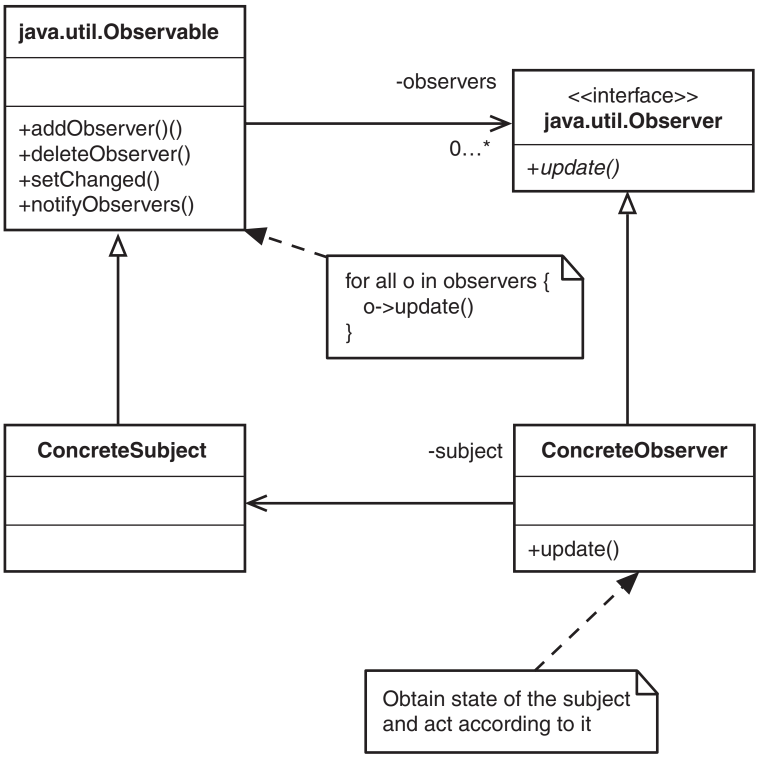


FIGURE 8.6 Observer pattern in Java

To achieve greater flexibility, the Broadcasters pattern encourages the use of scripts for implementation of observers. With this approach, we have scripts (rather than compiled components) that the subject registers and evaluates when a certain event occurs (or when it changes its state).

A traditional approach is to place a reference to the language interpreter and register scripts (through their location) to the subject. But of course, as was the case with the Mediator pattern, in the Java platform (and associated scripting languages), we can implement observers as Java classes in a scripting language that supports this functionality. In that way, we have both the good object-oriented design of Java systems and the additional flexibility of scripting.

Consequences

The Observer pattern introduces the following consequences.

- **Broadcast communication**—The subject does not care how many observers are currently registered. Observers can be added or removed at any time, and it is the observers' responsibility to decide whether they will react to a certain notification.
- **Unexpected updates**—Because observers are not aware of each other, changing a subject in the update process could result in unexpected update phases that are hard to track down.

Additionally, the Broadcasters pattern implies the following

- The behavior of observers can be easily modified without the need to rebuild the whole project. It can even be done at runtime.
- This pattern introduces additional runtime performance overhead because observers are interpreted every time an event occurs.

Sample Code

Let's start again with an implementation of our introductory example in Java (using the Observer pattern). Then we make the

necessary changes to enable the use of scripting languages (Groovy in this case) to implement our observers.

First, we create the subject component. As we already said, we have to extend the `java.util.Observable` class to enable multiple observers to be registered to the component (see Listing 8.12).

Listing 8.12 Subject Component

```
package net.scriptinginjava.ch8.observer;

import java.util.Observable;

public class Price extends Observable {

     Float price;

     public Float getPrice() {
         return price;
     }

     public void setPrice(Float price) {
         this.price = price;
         this.setChanged();
         this.notifyObservers();
     }

     public void increase() {
       setPrice(new Float(price.floatValue() + 1.0));
     }
}
```

This simple `price` object is just a wrapper around the float value for the price. You can see that we used two additional methods (inherited from the `Observable` class) in the `setPrice()` method. Those methods are used to notify registered observers that the price value has been changed.

The `setChanged()` method marks that the subject's state has been changed. The `notifyObservers()` method notifies all observers if the subject is marked as changed (we see how it does this in a moment).

Now that we have the subject, it is time to create the observers. First, we create an abstract `Label` class that will be a parent to all of our observer labels (see Listing 8.13).

Listing 8.13 Observer Label Widget

```
package net.scriptinginjava.ch8.observer;

import java.util.Observer;

import javax.swing.JLabel;

public abstract class Label extends JLabel implements Observer {

    public Label(String text) {
        super(text);
    }
}
```

The `Observable` interface defines only one method:

```
public void update(Observable o, Object arg);
```

That method must be implemented by all ancestors of this abstract `Label`. The `update()` method is called by the subject in the `updateObservers()` method. As you can see, the subject itself is passed to the observer.

If the object is passed to the subject's `updateObservers()` method, it will be passed as the second argument of the `update()` method of the registered observers. This additional object could help the observer to discover what has been changed in the subject. In this simple example, we don't need this extra object, because the price has only one field that could be changed.

Now we can implement observer labels (see Listing 8.14).

Listing 8.14 Price Observer Label Widget

```
package net.scriptinginjava.ch8.observer;

import java.util.Observable;

public class PriceLabel extends Label {

    public PriceLabel(String text) {
        super(text);
    }

    public void update(Observable o, Object arg) {
        this.setText(((Price)o).getPrice().toString());
    }
}
```

The PriceLabel class shows the total amount of the subject's Price object. On every Price change, the update() method is called, and the text in the label is updated according to this change (see Listing 8.15).

Listing 8.15 Tax Observer Label Widget

```
package net.scriptinginjava.ch8.observer;

import java.math.BigDecimal;
import java.util.Observable;

public class TaxLabel extends Label {

     private double tax = 0.175;

     public TaxLabel(String text) {
          super(text);
     }

     public void update(Observable o, Object arg) {
          double VAT = ((Price)o).getPrice().floatValue() * tax;
          int decimalPlace = 2;
          BigDecimal bd = new BigDecimal(VAT);
          bd = bd.setScale(decimalPlace,BigDecimal.ROUND_HALF_UP);
          VAT = bd.doubleValue();
          this.setText(new Double(VAT).toString());
     }
}
```

The TaxLabel class in this code is pretty much the same as the one in Listing 8.14. The only difference is that this label displays only the portion of the total price amount (17.5% in this example). The additional code is used to round the float value and to display only two decimal places.

The point here is that both labels are going to change their text when the price changes its value. But for that to happen, we have to instantiate all necessary objects first and then register these observers to the Price subject (see Listing 8.16).

Listing 8.16 Observer Client

```
package net.scriptinginjava.ch8.observer;

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

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

Listing 8.16 Continued

```
public class ObserverApp implements ActionListener {

     Price price;
     PriceLabel priceLabel;
     TaxLabel taxLabel;

     ObserverApp() {

          JFrame frame = new JFrame("Application");
          frame.getContentPane().setLayout(
               new FlowLayout()
          );
          frame.setSize( 150, 100 );

          JButton button = new JButton("Increase Price");
          button.addActionListener(this);
          frame.add(button);


          price = new Price();
          priceLabel = new PriceLabel("0");
          taxLabel = new TaxLabel("0");

          price.addObserver(priceLabel);
          price.addObserver(taxLabel);

          price.setPrice(new Float(50.00));

          frame.add(priceLabel);
          frame.add(taxLabel);


          frame.show();


     }

     public static void main(String[] args) {

          new ObserverApp();

     }

     public void actionPerformed(ActionEvent e) {

          price.increase();

     }
}
```

This application creates a frame with two labels (`PriceLabel` and `TaxLabel`) and a button. Note the code marked in bold in Listing 8.16. That code snippet instantiates a subject and two observers. It also registers those observers using the `addObserver()` method. This method is defined in the `Observable` class (extended by all subjects).

Of course, the `Observable` class defines some additional methods intended for observer handling. So, for example, the `deleteObserver()` and `deleteObservers()` methods are used to deregister observers from the subject. Also, with the `countObservers()` method, you can get the actual number of registered observers for a certain subject.

When the application is started, a frame like the one shown in Figure 8.7 appears.

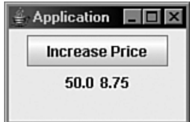


FIGURE 8.7 Observer example

Every time you click the button, the price will be increased (see the `increase()` method of the `Price` class). You can see that both labels are updated correctly through their `update()` methods (see Figure 8.8).

FIGURE 8.8 Observer example—price modified

To demonstrate the Broadcasters pattern, we can rename the `TaxLabel.java` file as `TaxLabel.groovy` and use it as a scripted component. Remember that you should check that the Java class is valid Groovy code before trying to use it in this way. In this example, there is nothing to stop us from using the previously defined Java class as the Groovy code.

We can now change our application to use a scripted observer for this purpose (see Listing 8.17).

Listing 8.17 Broadcasters Example

```
package net.scriptinginjava.ch8.observer;

import groovy.lang.GroovyClassLoader;
```

Listing 8.17 Continued

```
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import java.lang.reflect.Constructor;

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

public class ScriptApp implements ActionListener {

     Price price;
     PriceLabel priceLabel;
     Label taxLabel;

     ScriptApp() throws Exception {

          JFrame frame = new JFrame("Application");
          frame.getContentPane().setLayout( new FlowLayout() );
          frame.setSize( 150, 100 );

          JButton button = new JButton("Increase Price");
          button.addActionListener(this);
          frame.add(button);


          price = new Price();
          priceLabel = new PriceLabel("0");

          GroovyClassLoader loader = new GroovyClassLoader();
          Class observerClass = loader.parseClass(
               new File(
                    "net/scriptinginjava/ch8/"
                  + "observer/TaxLabel.groovy"
               )
           );
          Constructor observerConstructor =
               observerClass.getConstructor(
                    new Class[]{String.class}
               );
          taxLabel = (Label)observerConstructor.newInstance(
               new Object[] {"0"}
          );

          frame.add(priceLabel);
          frame.add(taxLabel);

          price.addObserver(priceLabel);
          price.addObserver(taxLabel);

          price.setPrice(new Float(50.00));

          frame.show();


     }

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

Listing 8.17 Continued

```
        new ScriptApp();

    }

    public void actionPerformed(ActionEvent e) {

        price.increase();

    }
}
```

The only difference is that now the `taxLabel` observer is evaluated and loaded from the Groovy script. This means it could be changed while the application is running, and that those changes will be reflected in the application. Because the main logic is located in the observers (when this pattern is used), we have achieved our goal of creating a flexible development environment in which the behavior is changeable during runtime.

> **Note**
> I left the second observer as the plain Java class for better readability of the example code.

Related Patterns

This pattern introduces an additional runtime performance overhead because observers are interpreted every time the event occurs. This problem could be solved with the Script Object Factory pattern.

Extension Point Pattern

This pattern explains how to extend components' behavior with simple-to-write scripts.

Problem

In Chapter 2, "Appropriate Applications for Scripting Languages," we discussed how important it is to be able to embed the scripting interpreter in a surrounding system-programming environment, such as JVM. One crucial reason for this is to enable developers to customize and extend the behavior of their components with scripts.

In our discussion of the Mediator and Observer patterns, I said that there are two ways in which we can make Java components use scripts. The first way, which we used in those examples, is to implement Java interfaces in scripting languages such as Groovy. The classes can easily be loaded through the specialized class loader that comes with the appropriate script engine. After those classes are loaded, they can be used further on, just as though they were regular Java classes. This method is important when you want to use a design-through-interface approach to constructing your system, and when you want to be able to modify the implementations of your interfaces during runtime.

The second approach, which is covered in the Extension Point pattern, is to let the component evaluate scripts from its methods. In this way, we can define the extension points of our components, and we can inject scripts into these points that can modify the behavior and the state of the component.

Solution

For the second approach to work, the component has to hold an instance of the language interpreter, instead of loading a class from the script file. We also have to provide the mechanism for setting the resource from which the script will be loaded.

Additionally, the component creator has to bind the component to the script to provide the component's context to script developers. Of course, other application classes could be bound to the script to make a richer context for its execution.

Consequences

This pattern introduces the following consequences.

- Inexperienced programmers can easily customize the component's behavior because the syntax of most scripting languages is easy to learn.
- System architects and component writers have to take special care regarding security. Security permissions must be defined for the script to prevent malicious activities from occurring. You can find more information about security for Groovy scripts in Chapter 4.

Sample Code

To demonstrate this pattern, we use a component whose behavior is extensible through Groovy scripts. Take a look at Listing 8.18.

Listing 8.18 Extension Point Example

```
package net.scriptinginjava.ch8;

import groovy.lang.Binding;
import groovy.lang.GroovyShell;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.StringWriter;

public class Component {

     private String script;
     private String someProperty;

     public String getScript() {
          return script;
     }
     public void setScript(String script) {
          this.script = script;
     }

     public void doSomeAction() throws Exception {
          if (script != null) {
               Binding binding = new Binding();
               binding.setVariable("comp", this);
               GroovyShell shell = new GroovyShell(binding);
               shell.evaluate(script);
          }

          // ... continue
     }

     public String getSomeProperty() {
          return someProperty;
     }

     public void setSomeProperty(String property) {
          this.someProperty = property;
     }

     public static void main(String[] args) throws Exception {
          Component component = new Component();
          component.setScript("println 'Hello world!'");
          component.doSomeAction();
     }
}
```

The `Component` class contains the `script` property, along with its *getter* and *setter* methods. Also, we defined the `doSomeAction()` method that should contain some business logic for this component. As you can see, this method evaluates the script, located in the property (if it has been set), before it proceeds to other logic. Note that the component is passed to the scripting engine context as the `comp` variable, so the script could use it to get or set some data. For example, it can work with the `someProperty` property defined in the component.

In this example, we just set the following simple script to be executed:

```
print "Hello world!"
```

As a result of its execution, you can expect that the following text will appear on the screen:

```
Hello world!
```

We can make this example more flexible if we enable component users to define scripts in the file (see Listing 8.19).

Listing 8.19 Modified Extension Point Example

```
package net.scriptinginjava.ch8;

import groovy.lang.Binding;
import groovy.lang.GroovyShell;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.StringWriter;

public class Component {

     private String script;
     private String someProperty;

     public String getScript() {
         return script;
     }
     public void setScript(String script) {
         this.script = script;
     }
```

Listing 8.19 Continued

```
    public void setScript(File file) throws IOException {
        BufferedReader bufIn = new BufferedReader(
            new FileReader(file)
        );
        StringWriter swOut = new StringWriter();
        PrintWriter pwOut = new PrintWriter(swOut);
        String tempLine;

        while ((tempLine = bufIn.readLine()) != null) {
          pwOut.println(tempLine);
        }

        pwOut.flush();
        setScript(swOut.toString());
    }

    public void doSomeAction() throws Exception {
        if (script != null) {
            Binding binding = new Binding();
            binding.setVariable("comp", this);
            GroovyShell shell = new GroovyShell(binding);
            shell.evaluate(script);
        }

        // ... continue the work
        System.out.println(someProperty);
    }

    public String getSomeProperty() {
        return someProperty;
    }

    public void setSomeProperty(String property) {
        this.someProperty = property;
    }

    public static void main(String[] args) throws Exception {
        Component component = new Component();
        component.setScript(
            new File(
                "net/scriptinginjava/ch8/componentInit.groovy"
            )
        );
        component.doSomeAction();
    }
}
```

We added the `setScript()` method that reads the submitted file and initializes the `script` property with its content. The code of this method is practically the same as that defined in the `getStringFromReader()` method of the `org.apache.bsf.util.IOUtils` class that we used in our BSF examples in Chapter 6, "Bean Scripting Framework."

In the main() method, we initialized the component with the componentInit.groovy file rather than the simple string script from Listing 8.18. Note, for example, that the componentInit.groovy file contains the code shown in the following code snippet:

```
println "Hello world!"
comp.someProperty = "value"
```

We used the comp variable in the script, which represents an instance of the component that evaluated the script. We also set the value for the someProperty property. Because we print its value in the doSomeAction() method, you can expect the following result to show up after executing this example:

```
Hello world!
value
```

The first line is printed from the script, and the second one is printed from the component's doSomeAction() method after the script has been evaluated.

You could also use this pattern with various Inversion of Control (IoC) containers, such as Spring (www.springframework.org), to initialize and extend the component behavior.

To summarize, this is one of the scripting patterns that is the basis for other patterns you can apply in the scripting environment. Its alternative is to implement Java classes in the scripting language of your choice and load them at runtime. Which method you use depends a great deal on your project's requirements and your personal affinities.

Related Patterns

You can use this pattern for implementing various scripting patterns. For example, you can use it to implement the Mediator and Observer patterns. This pattern also enables you to create various kinds of scripting interceptors.

Active File Pattern

This pattern explains how to use scripting to store both data and the code that handles that data in the file.

Problem

In recent years, we have witnessed a great general effort made to connect applications and systems built on different platforms and technologies. Such efforts placed great emphasis on the format of the data being transferred between systems.

The most important characteristic of those data formats is that they should be easily read and parsed in both scripting and system-programming languages. Because of that, XML is now the most widely accepted data format used for this purpose.

But in some situations, developers need even more flexibility, such as the ability to represent the same data in various formats or to perform certain data transformations before processing.

Solution

In Chapter 1, "Introduction to Scripting," I mentioned that code and data are interchangeable in most scripting languages. With this in mind, we can force our scripting application not only to store data in the file but also to create an executable script that contains both data and code. By applying the Active File pattern, we can actually run a script that can format data according to our needs (or do any other data transformation that we need at the moment).

Consequences

The Active File pattern introduces the following consequences.

- The application must embed a scripting interpreter to process data files like these.
- The file format could be too tightly coupled to the application. A developer could be tempted to use a data

format optimized for the specific problem. Instead, it is better to use widely used data formats, such as XML or CSV, for example, and thus enable data processing in other applications.

- Security must be taken into consideration when you are deciding whether to use this pattern. Because the application is executing a script that could possibly contain malicious code, a developer must accept scripts only from trusted sources and create a solution that would prevent any illegal activities by the script.

Sample Code

To demonstrate this pattern, let's consider the following example. We want to save a list of users, with their account balances, from our database into a file. Additionally, we want this data to be used in three formats:

- XML—As noted earlier, this is the most convenient format for processing data using some arbitrary application.
- **CSV (comma-separated values)**—This is convenient for importing data into various applications, such as spreadsheet applications, for example.
- SQL—You could use this to import data into a relational database for further processing.

Now take a look at the Groovy script (`active.groovy`) shown in Listing 8.20.

Listing 8.20 Active File Example

```
builder = new NodeBuilder()

// data

root = builder.users() {
     user(username:"mike", balance:100)
     user(username:"joe", balance:120)
}

// code

users = root.get("user");
```

Listing 8.20 Continued

```
def exportAsSQL(users) {
    users.each() {
        println “””INSERT INTO USERS (username, balance) \
                    VALUES ( ‘${it.attribute(“username”)}’
                    , ${it.attribute(“balance”)} );”””;
    }
}

def exportAsCSV(users) {
    users.each() {
        println “””${it.attribute(“username”)} \
                   ,${it.attribute(“balance”)} “””;
    }
}

def exportAsXML(users) {
    println “<users>”;
    users.each() {
        println “””<user \
                    username = “${it.attribute(“username”)}”> \
                    ${it.attribute(“balance”)}</user>”””;
    }
    println “</users>”;
}

if (args.size() == 0)
    action = “csv”
else
    action = args[0]


switch (action) {
    case “sql” : exportAsSQL(users)
            break;
    case “xml” : exportAsXML(users)
            break;
    default : exportAsCSV(users)
}
```

At the beginning of the script, we defined the data in a neutral treelike structure. We used Groovy’s `groovy.util.NodeBuilder` class for this task (see Chapter 5, “Advanced Groovy Programming,” for more details about this class).

The rest of the file contains the code that can format this data in one of the three formats we specified earlier. For that purpose, we defined three functions: `exportAsSQL()`, `exportAsXML()`, and `exportAsCSV()`. Which one is called depends on the first command-line argument that is passed to the script. If no arguments are passed, the CSV format will be assumed.

So if you run the script with the following command:

```
groovy active.groovy
```

the `exportAsCSV()` method will be called, and you will get the following output on the console:

```
mike,100
joe,120
```

You would get the same result if you ran the script with the following command:

```
groovy active.groovy csv
```

If you like to get data in XML format, just run the same script and pass `xml` as an argument to it:

```
groovy active.groovy xml
```

The result will look like this:

```
<users>
<user username = “mike”>100</user>
<user username = “joe”>120</user>
</users>
```

At the end, this script call

```
groovy active.groovy sql
```

will result in the following output:

```
INSERT INTO USERS (username, balance) VALUES ( ‘mike’, 100
);
INSERT INTO USERS (username, balance) VALUES ( ‘joe’, 120
);
```

The next natural question is can we easily make this kind of active file with Groovy? Fortunately, the answer is yes. We use Groovy templates, described in Chapter 5, to do this. We can start by creating a template, as shown in Listing 8.21.

Listing 8.21 Active File Template

```
builder = new NodeBuilder()

root = builder.users() {
<%
    users.each() {
        out.println("user(username:'${it.username}' \
                      , balance:${it.balance})");
    }
%>
}

users = root.get("user");

def exportAsSQL(users) {
    users.each() {
        println """INSERT INTO USERS (username, balance) \
                   VALUES ( '\${it.attribute("username")}' \
                   , \${it.attribute("balance")} );""";
    }
}

def exportAsCSV(users) {
    users.each() {
        println """\${it.attribute("username")} \
                  ,\${it.attribute("balance")} """;
    }
}

def exportAsXML(users) {
    println "<users>";
    users.each() {
        println """<user username = \
                   "\${it.attribute("username")}"> \
                    \${it.attribute("balance")}</user>""";
    }
    println "</users>";
}

if (args.size() == 0)
    action = "csv"
else
    action = args[0]


switch (action) {
    case "sql" : exportAsSQL(users)
            break;
    case "xml" : exportAsXML(users)
            break;
    default : exportAsCSV(users)
}
```

As you can see, this template is practically the same as the active file we want to create. The only difference is the code that is marked in bold. That code uses the list named `users` to populate concrete data into the file.

All that we need now is a Groovy script that will get data from the source (a relational database, for example) and make an active file from this template (see Listing 8.22).

Listing 8.22 Active File Generator

```
import groovy.sql.Sql
import groovy.text.Template
import groovy.text.TemplateEngine
import groovy.text.SimpleTemplateEngine
import java.io.File
import java.io.FileWriter

sql = Sql.newInstance("jdbc:mysql://localhost/groovy"
           , "com.mysql.jdbc.Driver")

users = []

sql.eachRow("SELECT * FROM users") {
    users << it
}

TemplateEngine engine = new SimpleTemplateEngine()
Template template = engine.createTemplate(
    new File("net\\scriptinginjava\\ch8\\activeTemplate")
)
result = template.make(users:users)

fileWriter = new FileWriter("activetest.groovy")
result.writeTo(fileWriter)
```

In this Groovy script, we used GroovySQL to get all the user data from our `users` table. We created a list named `users` from it and passed it to the template. As a result, we created an `activetest.groovy` script that is equivalent in behavior to the active file described at the beginning of this section.

This pattern gave us an opportunity to store data with additional logic that could be used to easily represent that data in various forms. The client could also use this logic to make various transformations on this data.

Conclusion

This chapter provided some useful patterns for mixing Java and scripting code. The patterns presented here by no means represent all the possible solutions to this type of problem. Instead, they represent a small set of useful ideas that are applicable in

this context. Many other patterns are a natural extension of the ones presented in this chapter.

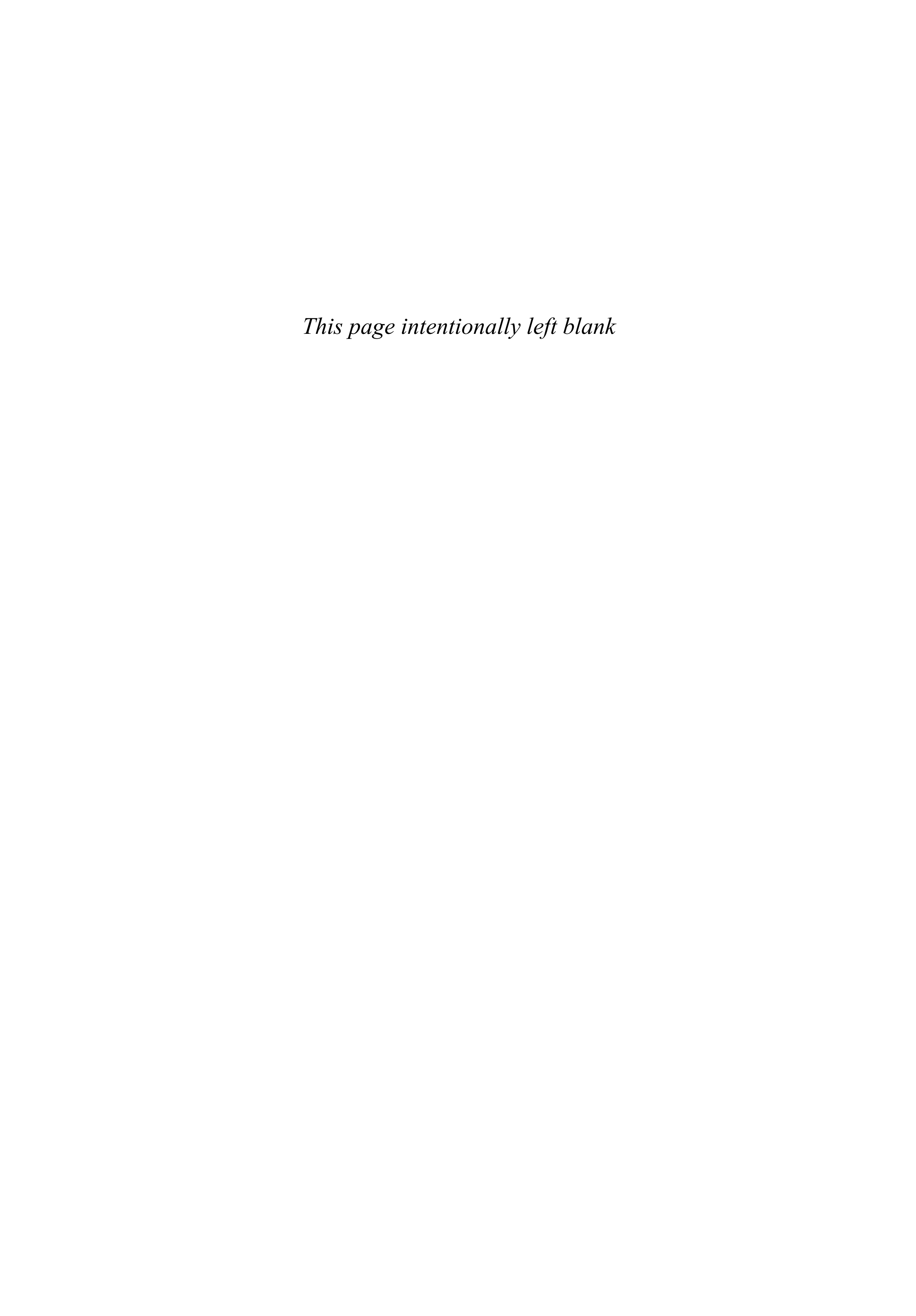
This material should be just an entry point to this topic, and I hope that it will prompt you to create more patterns that could be useful in this domain of software development. You can find some additional scripting patterns in Nat Pryce's collection, which he has posted on his Web site, www.doc.ic.ac.uk/~np2/patterns/scripting/.

Before we move on to the next chapter, I want to say a few words about another side to design patterns. Pattern critics often point out that inexperienced developers could fall into the *pattern trap*. This term is used to describe a situation where a developer tries to implement as many patterns as possible in his application, regardless of whether they are really applicable in the context he is working in.

You need to satisfy three criteria to be successful in using a pattern to solve a certain problem:

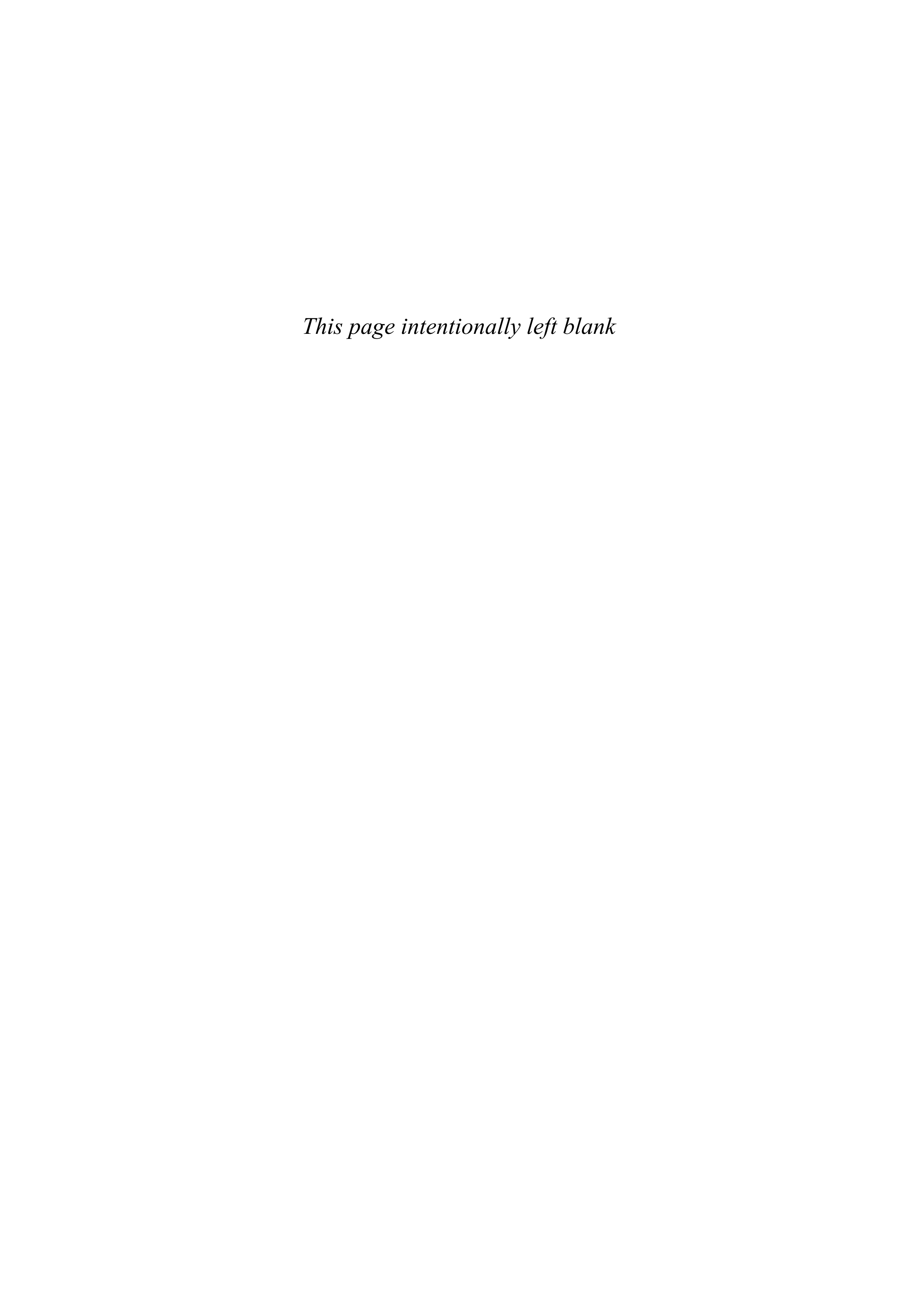
- Understand the problem.
- Understand the pattern.
- Understand how the pattern solves the problem.

Although this seems natural, be sure that all three criteria are satisfied before you apply a pattern. Using a pattern just for the sake of saying that you did could, in the end, produce more problems than it solves.


This page intentionally left blank

CHAPTER 9 Scripting API

CHAPTER 10 Web Scripting Framework



This page intentionally left blank

Chapter 9

Scripting API

Back in Chapter 6, "Bean Scripting Framework," I described the BSF library and explained the need for a general scripting framework like it. Although the BSF is a solid and stable project that serves its purpose well, the Java community needs a solution that better suits modern scripting languages and integrates more easily with native interpreters. In this chapter, I describe the Scripting API, a specification that arose from the Java Community Process (JCP) to create a standard Java scripting framework for the Java platform.

Here, we explore the motivation behind this API and its abstractions. We also walk through numerous examples that show us how to use these abstractions to solve everyday programming problems.

Motivation and History

In Chapter 2, “Appropriate Applications for Scripting Languages,” I talked about the use of scripting languages for developing dynamic Web applications. I cited PHP as one of the most popular languages for that task. Proof of its popularity exists in the crucial role it plays in the LAMP (Linux, Apache, MySQL, PHP) platform. The combination of these four technologies was proven by many successful deployments to be a stable and powerful platform for building dynamic Web applications.

PHP has a large community of software developers and has become a de facto choice for implementation of small and midsize Web solutions. The main advantage of PHP is its easy-to-learn syntax and concepts, which allow even novice programmers a moderate learning curve. Also, its architecture is flexible enough to enable the creation of a large number of language extensions that allow application developers to do almost anything they can think of.

Over time, PHP projects have grown larger, and people have wanted to develop more complex applications with it. However, PHP was not originally designed for such tasks. It has a page-centric architecture, and until Version 5, it had limited support for object-oriented programming. Furthermore, there is no application container in PHP, so all variables are bound to request or session scopes. This lack of an application container implies the lack of system support for transaction management, caching, software components, and all the other things developers of enterprise applications are used to having.

As applications get larger and more complex, scalability becomes a new issue. PHP is not as scalable as other enterprise development tools are, and there is no easy and standard way to cluster it for better performance.

For all these reasons, PHP applications (with respectable exceptions) remained dominant primarily in the area of small and midsize dynamic Web applications, with modest applications in enterprise development.

Java, on the other hand, had a different development path. It was originally designed as an object-oriented language for

developing client applications that could be easily transferred through a network. Because of good language concepts and true platform independency, many people saw tremendous possibilities for Java. Thus, three Java platforms emerged:

- **J2SE**—This is the Java Standard Edition, which contains core functionality for development of client and server applications (including Web applications).
- **J2ME**—This is the Java Micro Edition, which is targeted for development of embedded applications for mobile phones and similar devices.
- **J2EE**—This is the Java Enterprise Edition, which defines a standard for development of large, component-based enterprise applications.

Java quickly became a popular language of choice for development of enterprise and Web applications. All the features that PHP does not include are available to Java developers through the Servlet specification and the Enterprise Edition. Application servers that comply with the J2EE specification implement support for software components (EJBs), declarative transaction management, declarative security, integration with legacy systems, and many other features needed in this problem domain. Also, Java is much more suitable for clustering and distributed computing than scripting languages that are tightly coupled to the Web server environment. All this makes Java a good choice as a development platform for large and complex server applications.

The main problem with this platform was its complexity and the fact that Java development is time consuming. Even if we take an ordinary Web application that does not use J2EE features, a developer needs to know many technologies (such as JSP, the Servlet specification, XML, and so on) just to begin development. And after development begins, it takes much more time to produce desired functionalities than it would with the LAMP platform.

Many frameworks that force the Model-View-Controller (MVC) pattern, template engines, Object Relational mapping tools, IoC containers, and similar technologies emerged to

address this problem. Nevertheless, the aforementioned complexity, and the fact that some projects missed their deadlines, remained the major drawback of the Java platform.

Considering the large community of PHP developers and the huge code base of Web solutions for many problem domains on the one hand, and the power of the Java platform (with its community backed by large corporations) on the other, it is not surprising that ideas and initiatives for integrating these two platforms were born at Sun Microsystems (Java's side) and Zend Technologies (the PHP company).

Prior to the availability of the Scripting API, solutions for integrating Java and PHP applications were, to put it mildly, modest. PHP has a mechanism that you can use to integrate it into a servlet container using the SAPI module and to expose the Java Virtual Machine (JVM) to applications, but this solution was not reliable enough for use in mission-critical tasks. There is no support for this extension and no accurate documentation. The PHP manual that covers Java integration (www.php.net/manual/en/ref.java.php) describes this problem well:

This extension is EXPERIMENTAL. Use this extension at your own risk.

Another way to perform this integration is through Web services and the XML-RPC protocol. This approach works great in some applications, but it is not applicable in all cases. Also, performance issues of the protocol make this solution unacceptable in many cases.

The Scripting API fills this gap by providing a standard solution for integrating scripting engines and the JVM. Also, the fact that this API is an integral part of the Java platform should guarantee its wide acceptance among developers.

Introduction

The initiative for bringing Java and PHP together is in the Java Specification Request 223 (JSR 223; www.jcp.org/en/jsr/detail?id=223). It was originally called "Scripting Pages in Java

Web Applications" and its goal was to provide a standard way to generate Web content in scripting languages. PHP was chosen as a reference scripting language for this specification.

Unlike other technologies already covered in this book, this specification was not focused only on scripting languages that have interpreters implemented in Java. Integration of those languages is much easier to achieve than it is with native scripting languages, such as PHP.

So to meet the original specification's goal, it was necessary to create a framework, similar to the BSF (described in Chapter 6), that provides common abstractions (interfaces and classes) that you can use to integrate Java and arbitrary scripting languages.

Therefore, the specification was renamed to "Scripting for the Java Platform," and its original goal was extended with the specification of such an API. So, the specification contained two main elements:

- **General Scripting API**—This is a framework that allows you to embed scripting engines (both Java and native ones) into Java applications. One of the main design goals of this API was to provide backward compatibility with existing frameworks, such as the BSF, to make transition of projects and adoption by developers as easy as possible.
- **Web Scripting API**—This uses the Scripting API to allow you to embed scripting engines into servlet containers. The framework allows pages written in scripting languages, such as PHP, to be included in Java Web applications. It also defines mechanisms for resource sharing among scripts and other elements of Web applications (servlets, JSP pages, and so on).

The Web Scripting API is removed from the final version of the specification because it will be subject to further development. But still, we describe it in this book because its concepts represent the basics of Java and scripting integration in the Web environment. This API could be a starting point for any future work in this field.

The specification does not define the syntax that the scripting language should implement to use Java objects. Instead, it includes a discussion of "Java Language Bindings," which covers mechanisms that you can use by specification implementations to map script method calls to Java object method calls. It also defines a mechanism for conversion of values between Java objects and scripting language variables.

This chapter covers the Scripting API. I describe all the concepts and abstractions that it introduces through appropriate examples and highlight some of the differences between this API and the BSF.

Chapter 10, "Web Scripting Framework," covers the Web Scripting Framework and its applicability in day-to-day development tasks.

Getting Started

The Scripting API's interfaces and classes are located in the `javax.script` package, and the Web Scripting Framework files are in the `javax.script.http` package. The Scripting API (`javax.script` package) is an integral part of the Java Standard Edition with the Mustang release (J2SE 6.0) onward.

If you want to use the Scripting API with Java releases prior to this one, you can find instructions on how to obtain and install the Reference Implementation (RI) of this specification in Appendix C, "Installing JSR 223." Note that the API included in Java 6 JDK and the Reference Implementation differs in some segments. In this chapter, we use a strictly JDK 6 compatible API, so if you want to run examples, you need to install Java JDK 6 or newer.

Sun Java SE 6 JDK includes scripting engine support for JavaScript Rhino implementation described in Chapter 3, "Scripting Languages Inside the JVM." In the RI, you can find implementations of PHP, Rhino, and Groovy scripting engines. Finally, there is a *Scripting* project (https://scripting.dev.java.net/) whose purpose is to provide engine implementations for most scripting languages available today. There you should start looking for engine implementation for your favorite scripting

language. In this chapter, examples are written mostly in JavaScript, but there are a few Groovy examples too. So, if you want to run these examples, make sure that you obtain the Scripting project distribution and include the Groovy engine implementation in the classpath.

Architecture

I already said that one of the main design goals of this API was to provide backward compatibility with existing interfaces used for this kind of task. (By *existing interfaces*, I mean the interfaces provided by the scripting engine implementations covered in Chapters 3 and 4, and the BSF API, which is the general-purpose library that has been used until now, described in Chapter 6.)

One more important design goal of this API is portability. It is meant to be a standard API for embedding all kinds of scripting languages. Scripting languages vary a great deal in terms of the functionalities they provide. This API tends to cover all the features that a certain scripting engine can provide, but at the same time, it must enable simple engines, with just basic functionalities, to comply with the API.

Therefore, the Scripting API provides application developers the ability to determine features that are implemented in certain scripting engines at runtime. In that way, developers can adjust their code for specific cases, or fail correctly if the scripting engine does not implement certain optional features. We see how these portability goals are implemented later in this chapter.

First, I discuss the abstractions and concepts defined in the API and explain their purpose through examples.

Discovery Mechanism

As in our discussion of the BSF API, I start by explaining the `javax.script.ScriptEngineManager` class. The purpose of this class is similar to that of the `BSFManager` class: It serves as the general registry of available scripting engines.

BSF has a simple static language discovery mechanism used to get all languages currently registered with the manager. It performs this task through the `Languages.properties` file located in the root library package.

The Scripting API approaches this problem differently. It is based on the service provider mechanism described in the Jar File Specification. According to this specification, the service is a set of interfaces and (possibly abstract) classes. The service provider represents an implementation of the service (an implementation of its interfaces and abstract classes).

This mechanism should allow you to make service providers available to the application dynamically, by adding them to the classpath. For that purpose, the Jar File Specification specifies that files located in the `META-INF/services` folder of the JAR archives should be used as the service providers' configuration files. Furthermore, the configuration files should be named after the service interfaces (or abstract classes) they implement, and the name must include the service package as well. Finally, the files should contain a newline-separated list of particular classes that implement that service.

As we see in a moment, scripting engines are created through the factory method defined in the `javax.script.ScriptEngineFactory` interface. So the `ScriptEngineManager` class searches through all the JAR files in the application's classpath and registers engine factories that are found in the `META-INF/services/javax.script.ScriptEngineFactory` files of those archives. You can find an example of this file in the JDK's `lib/resources.jar` file. It looks like this:

```
#script engines supported

com.sun.script.javascript.RhinoScriptEngineFactory
#javascript
```

As you can see, only one implementation of the `ScriptEngineFactory` interface (in other words, one service provider) is defined in this JAR file. By its name and comments, we can tell that it is the factory for the engine of the JavaScript

language. Now we can write an example that prints names of all registered languages (see Listing 9.1).

Listing 9.1 Discovery Mechanism

```
package net.scriptinginjava.ch9;

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


public class ManagerTest {

    public static void main(String[] args) {

        ScriptEngineManager manager =
            new ScriptEngineManager();
        List<ScriptEngineFactory> factories =
            manager.getEngineFactories();
        System.out.println("Available languages:");
        for (ScriptEngineFactory factory: factories) {
            System.out.println(factory.getLanguageName());
        }

    }
}
```

Now if we execute this code sample, it should produce the following output:

```
Available languages:
ECMAScript
```

When instantiated, the manager looks for all engine factory implementations so that the `getEngineFactories()` method will return them in a list. The `getLanguageName()` method of the `factory` object, as I explain in more detail in the following sections, returns the name of the language for which it creates engine objects.

Engine Metadata

As you saw in the preceding section, engine metadata is not defined in the discovery mechanism configuration files, as was the case in the BSF. Therefore, the Scripting API defines methods in the `ScriptEngineFactory` interface that is used to describe engines created by that factory.

In Listing 9.1, we used the getLanguageName() method of the ScriptEngineFactory interface that returns the name of the language that factory produces. Listing 9.2 demonstrates the rest of these metadata methods. (I explain them afterward.)

Listing 9.2 Engine Metadata

```
package net.scriptinginjava.ch9;

import java.util.List;

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

public class InfoTest {

    public static void main(String[] args) {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngineFactory factory =
            manager.getEngineByName("js").getFactory();
        System.out.println("JS engine metadata");
        System.out.println(
            "Engine name: " + factory.getEngineName()
        );
        System.out.println(
            "Engine version: " + factory.getEngineVersion()
        );
        System.out.println(
            "Language name: " + factory.getLanguageName()
        );
        System.out.println(
            "Language version: " + factory.getLanguageVersion()
        );
        List<String> extensions = factory.getExtensions();
        System.out.print("Language extensions: ");
        for(String extension: extensions) {
            System.out.print(extension + " ");
        }
        System.out.println();
        System.out.print("Language mime types: ");
        List<String> mimeTypes = factory.getMimeTypes();
        for (String mimeType: mimeTypes) {
            System.out.print(mimeType + " ");
        }
        System.out.println();
        List<String> names = factory.getNames();
        System.out.print("Engine names: ");
        for (String name: names) {
            System.out.print(name + " ");
        }
        System.out.println();
    }
}
```

If we run this application against the JavaScript engine included in the JDK 6, we get the following output:

```
JS engine metadata
Engine name: Mozilla Rhino
Engine version: 1.6 release 2
Language name: ECMAScript
Language version: 1.6
Language extensions: js
Language mime types: application/javascript
  application/ecmascript text/javascript text/ecmascript
Engine names: js rhino JavaScript javascript ECMAScript
  ecmascript
```

The first four methods provide information about the names and versions of the language and the particular engine. You can use them to take appropriate action according to certain engine and language implementations.

The last three methods return lists that specify extensions, mime types, and names associated with the scripting language and engine.

Creating and Registering Scripting Engines

Just as in the BSF, the Scripting API defines an abstraction for scripting language interpreters. It defines this abstraction in the `javax.script.ScriptEngine` interface. We come back to this interface and related abstractions later in this chapter. First, we have to determine how we can create instances of particular classes that implement this interface.

I said that you can use the `factory` method of the `ScriptEngineFactory` interface to create an instance of the language interpreter (engine) for a desired language. Listing 9.3 demonstrates one approach that you can take to create a `ScriptEngine` instance through this method.

Listing 9.3 A `getScriptEngine()` Method Example

```
package net.scriptinginjava.ch9;

import java.util.List;

import javax.script.ScriptEngine;
```

Listing 9.3 Continued

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


public class CreateTest {

    public static void main(String[] args) {
        ScriptEngineManager manager = new ScriptEngineManager();
        List<ScriptEngineFactory> factories =
            manager.getEngineFactories();
        ScriptEngine engine = null;
        for (ScriptEngineFactory factory: factories) {
            if (factory.getLanguageName()
                    .equalsIgnoreCase("EcmaScript")
             ) {
                engine = factory.getScriptEngine();
                break;
            }
        }

        if (engine != null) {
            System.out.println("JS engine implemented by: "
                              + engine.getClass());
        } else {
            System.out.println("No engine found for JS");
        }
    }
}
```

In Listing 9.3, we loop through all the engines registered with the manager. If we find the scripting factory whose language name metadata equals ECMAScript—the ECMAScript engine, in other words—we will create an instance of that engine using the `getScriptEngine() factory` method. The example should print the following text on output if it is used with the Scripting API distributed with JDK 6:

```
JS engine implemented by: class
  com.sun.script.javascript.RhinoScriptEngine
```

Creation Methods

The previous lookup mechanism is not particularly convenient, so the specification adds a few methods to the `ScriptEngineManager` class for tasks like this. There are three ways to select an appropriate engine:

- By its name
- By its associated filename extension
- By its associated MIME type

To this end, three methods are defined:

- `getEngineByName()`
- `getEngineByExtension()`
- `getEngineByMimeType()`

getEngineByName()

This method is used to look through the available engine factories and create the appropriate engine if the factory for the specified language is found. So the code in Listing 9.4 is almost entirely equal in functionality to the example shown in Listing 9.3 (the difference is that this method will look up through all engine names, not only the main one).

Listing 9.4 A `getEngineByName()` Method Example

```
package net.scriptinginjava.ch9;

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

public class ByNameTest {

    public static void main(String[] args) {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("js");
        if (engine != null) {
            System.out.println("JS engine implemented by: "
                             + engine.getClass()
            );
        } else {
            System.out.println("No engine found for JS");
        }
    }
}
```

After executing Listing 9.4, you will get the same result as before. The only difference is that this manager's lookup method sets the context of the created engine (but that is the topic of later sections in this chapter). For now, suffice it to say that this is the case for all the lookup methods of the `ScriptEngineManager` class.

getEngineByExtension()

This method is used to look up the factory that creates the scripting engine capable of handling files with the given extension. If no such factory exists, it will return a `null` value. Look at Listing 9.5.

Listing 9.5 A `getEngineByExtension()` Method Example

```
package net.scriptinginjava.ch9;

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

public class ByExtensionTest {

     public static void main(String[] args) {
          ScriptEngineManager manager = new ScriptEngineManager();
          ScriptEngine engine =
               manager.getEngineByExtension("js");
          if (engine != null) {
               System.out.println("js files are handled by: "
                                   + engine.getClass() + " engine"
               );
          } else {
               System.out.println(
                    "No engine found for .js extension"
               );
          }
     }
}
```

We created a scripting engine implementation that can handle `.js` file extension scripts. This method could be helpful when you want to allow support for arbitrary scripting language files in your product.

getEngineByMimeType()

Because this library is also meant to be useful in a Web context, this method is used to make it easier to look up engines that can handle a certain mime type. It has practically the same syntax as the methods described in Listing 9.4 and Listing 9.5. So, I will skip its basic example and demonstrate this method later in the chapter through the advanced example.

Registration Methods

Along with these methods come supplementary methods used to register the engine factory, which handles a certain language, extension, or mime type. These methods are as follows:

- `registerEngineName()`
- `registerEngineExtension()`
- `registerEngineMimeType()`

I demonstrate these methods by registering the `com.sun.script.javascript.RhinoScriptEngine` engine to handle all the *text/xml* mime types (see Listing 9.6).

Listing 9.6 Register Engine Example

```
package net.scriptinginjava.ch9;

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

import com.sun.script.javascript.RhinoScriptEngineFactory;

public class RegisterTest {

    public static void main(String[] args) {
        ScriptEngineManager manager = new ScriptEngineManager();
        manager.registerEngineMimeType("text/xml"
            , new RhinoScriptEngineFactory());
        ScriptEngine engine =
            manager.getEngineByMimeType("text/xml");

        if (engine != null) {
            System.out.println(
                "XML resources will be handled by: "
               + engine.getClass() + " engine"
            );
        } else {
            System.out.println(
                "No engine found for handling XML resources"
            );
        }
    }
}
```

The call to these register methods will overload any such association found during the discovery mechanism.

Evaluation

Now that we know how to create an instance of the scripting engine for the desired language, it's time to see what it can do for us. The most important thing is surely the evaluation of scripts. This is done through the `eval()` method of the `ScriptEngine` interface, as shown in Listing 9.7.

Listing 9.7 Evaluating Script Contained in a String

```
package net.scriptinginjava.ch9;

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


public class EvalTest {

     public static void main(String[] args) {
          ScriptEngineManager manager = new ScriptEngineManager();
          ScriptEngine engine = manager.getEngineByName("js");
          if (engine != null) {
               try {
                    engine.eval("println('Hello world!')");
               } catch (ScriptException se) {
                    System.out.println(se.getMessage());
               }
          }
     }
}
```

In Listing 9.7, we instanced the manager, used it to create a Rhino engine, and evaluated the simple statement that prints the following text on standard output:

```
Hello world!
```

As you probably noticed, the `eval()` method that we used in Listing 9.7 takes a script as a `String` argument. One of the features of the Scripting API that is missing in the BSF is an `eval()` method that accepts a `Reader` argument. This method signature is used to evaluate scripts defined in files, streams, and other resources. There is no need anymore for the helper method to achieve this functionality.

To demonstrate this method, let’s define a simple JavaScript script named `hello.js`:

```
println(‘Hello world!’);
```

Now, we can evaluate it with the Java program shown in Listing 9.8.

Listing 9.8 Evaluating Script Contained in a File

```
package net.scriptinginjava.ch9;

import java.io.FileReader;
import java.io. FileNotFoundException;

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

public class EvalReaderTest {

    public static void main(String[] args) {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName(“js”);
        if (engine != null) {
            try {
                engine.eval(
                    new FileReader(
                        “net/scriptinginjava/ch9/hello.js”
                    )
                );
            } catch (ScriptException se) {
                System.out.println(se.getMessage());
            } catch (FileNotFoundException fnfe) {
                fnfe.printStackTrace();
            }
        }
    }
}
```

In Listing 9.8, we created a `java.io.FileReader` object for the script file and simply passed it to the `eval()` method. Of course, we had to catch `java.io.FileNotFoundException` because the `FileReader` constructor can throw it.

You may recall from our discussion in Chapter 6, “Bean Scripting Framework,” that the BSF API has distinct methods for evaluating scripts that return values and for those that don’t. The Scripting API does not make that distinction. The `eval()` method that I described here returns an `Object` result.

That result is the returned value from the script, if that scripting language supports that functionality and that script has returned some value, or the `null` value otherwise.

I demonstrate this functionality with the following script (`return.js`):

```
value = "JavaScript"
println("Returning value: " + value)
value
```

The script is simple. It prints out a line and returns a `JavaScript` value. Note that the value of the last line is returned by the script (in this case, it is the `value` variable). Now let's evaluate this script and handle the returned value (see Listing 9.9).

Listing 9.9 Handling Script Evaluation Result

```
package net.scriptinginjava.ch9;

import java.io.FileNotFoundException;
import java.io.FileReader;

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

public class ReturnTest {

     public static void main(String[] args) {
          ScriptEngineManager manager = new ScriptEngineManager();
          ScriptEngine engine = manager.getEngineByName("js");
          if (engine != null) {
               try {
                    Object result = engine.eval(
                         new FileReader(
                              "net/scriptinginjava/ch9/return.js"
                         )
                    );
                    System.out.println("Returned value: " + result);
               } catch (ScriptException se) {
                    System.out.println(se.getMessage());
               } catch (FileNotFoundException fnfe) {
                    fnfe.printStackTrace();
               }
          }
     }
}
```

As you can see, the only thing different in Listing 9.9 compared to Listing 9.8 is that now we collect the value returned by the `eval()` method and handle it further.

After execution, this Java program is expected to print the following text on standard output:

```
Returning value: JavaScript
Returned value: JavaScript
```

By specifying only one method for script evaluation, regardless of whether it should return a value, the Scripting API is made cleaner and easier to use.

ScriptException

You have probably noticed that the `eval()` method throws a `javax.script.ScriptException`. Like many other libraries that deal with scripting languages and interpreters, `ScriptException` has properties that can store some additional information about the error that occurred. The properties are the following:

- **`message`**—Can contain the message thrown by the language interpreter, which describes the error
- **`filename`**—Can contain the name of the file in which the script is located (if it is applicable)
- **`lineNumber`**—Can further narrow the location of the error by specifying the line number in the script
- **`columnNumber`**—Again, if populated, indicates the column number of the line where the error has occurred

Listing 9.10 demonstrates the possible use of these properties.

Listing 9.10 Handling `ScriptException`

```
package net.scriptinginjava.ch9;

import java.io.FileNotFoundException;
import java.io.FileReader;

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

Listing 9.10 Continued

```
import javax.script.ScriptException;

public class ExceptionTest {

    public static void main(String[] args) {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("js");
        if (engine != null) {
            try {
                engine.eval(
                    new FileReader(
                        "net/scriptinginjava/ch9/error.js"
                    )
                );
            } catch (ScriptException se) {
                System.out.println(
                        se.getMessage()
                        + " in file " + se.getFileName()
                        + " on line " + se.getLineNumber()
                        + " in column " + se.getColumnNumber()
                );
            } catch (FileNotFoundException fnfe) {
                fnfe.printStackTrace();
            }
        }
    }
}
```

Whether these properties will be populated depends on the scripting language engine implementation. The specification does not force engine implementations to do this, so you can't rely on them, but they could be valuable for debugging purposes.

Binding

For a framework like this, a crucial task is sharing data between host (Java) applications and scripting engines (scripts), so we are going to give it the attention it deserves.

We saw that the BSF library has implemented a simple, plain model of variable binding, meaning that shared variables are registered with the engine manager and then they are accessible by all engines created by that manager. The only decision you can make is whether you are going to register a variable when it is accessible through the central object repository, or declare it when it is mapped directly to the script variable. There

is no way to bind variables only to a certain engine, or to group them according to their purpose.

The Scripting API, on the other hand, is more careful in this area. Every script is executed in its context, which contains variables accessible by the script and some other state objects that are explained later in this chapter. The difference is that now those variables are grouped in *namespaces* or *scopes*. Namespaces are basically key/value pairs that bind variable values to their keys.

This abstraction is described by the `javax.script.Bindings` interface. This interface is an extension of the `java.util.Map<java.lang.String, java.lang.Object>` collection interface, which exactly matches the nature of namespaces.

I said that variables are grouped in namespaces (scopes), so the context in which a script is executed contains more than one namespace. The script context abstraction is defined in the `javax.script.ScriptContext` interface. One way you can think of this interface is as a set of namespaces exposed to the script through its scripting engine.

This specification defines two namespaces:

- **Engine scope**—The engine scope namespace holds the engine-specific data binding. Variables that are bound to one engine's scope are not visible in another engine, and vice versa.
- **Global scope**—The global scope namespace holds the manager's (or the application's) specific state. All variables bound to this scope are accessible by all engines created by that manager.

Now that you have a basic introduction to the binding concepts of the Scripting API, we will discuss all its elements in more detail, and through examples.

Engine Scope

As we discussed, the engine scope holds variable binding that is specific to a certain script engine. Other script engines created

by the same manager cannot access this data. The host application's Java objects that are put in this scope are visible like variables in the script.

Let's take, for example, the following JavaScript script:

```
println("Hello " + name);
```

We want to execute this script using the Scripting API, but we also want to define the value for the `name` variable first. We can use the Java application in Listing 9.11 for this task.

Listing 9.11 Binding Example

```
package net.scriptinginjava.ch9;

import java.io.FileReader;

import javax.script.ScriptContext;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.SimpleBindings;


public class BindingsTest {

    public static void main(String[] args) throws Exception {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("js");
        SimpleBindings bindings = new SimpleBindings();
        bindings.put("name", "Dejan");
        engine.setBindings(bindings
            , ScriptContext.ENGINE_SCOPE);
        engine.eval(
            new FileReader(
                "net/scriptinginjava/ch9/bindings.js"
            )
        );
    }
}
```

The code marked in bold in Listing 9.11 creates the desired data binding to the scripting engine's scope. First, we created an instance of the `SimpleBindings` class, which is the implementation of the `Bindings` interface. Because the `Bindings` interface is an extension of the `java.util.Map` interface, we used the `put()` method to create an association between the `name` key and its value. Finally, this binding is registered to the

scripting engine's context by the `setBindings()` method. The second argument in this method is actually the `int` type key that is used for accessing a desired namespace in the set of namespaces contained in the context.

If we execute this Java application, the `String` object with the value `Dejan` that is put in the engine's scope will be mapped to the `name` JavaScript variable. As a result, the evaluated script prints the following text on standard output:

```
Hello Dejan
```

OVERRIDING THE ENGINE SCOPE

The engine's scope can also be passed through the `eval()` method as the second argument, as shown in Listing 9.12.

Listing 9.12 Overriding the Engine Scope

```
package net.scriptinginjava.ch9;

import java.io.FileReader;

import javax.script.ScriptContext;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.SimpleBindings;

public class BindingsTest1 {

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

        SimpleBindings bindings = new SimpleBindings();
        bindings.put("name", "Dejan");

        engine.setBindings(bindings, ScriptContext.ENGINE_SCOPE);

        SimpleBindings evalBindings = new SimpleBindings();
        evalBindings.put("name", "Joe");

        engine.eval(
            new FileReader(
               "net/scriptinginjava/ch9/bindings.js"
            )
          , evalBindings
        );
    }
}
```

In this example, we created a script engine and initialized its namespace in the same way we did in Listing 9.11. But here, we created another namespace instance and passed it to the `eval()` method. By doing this, we have overridden the original engine's scope during script execution.

As a result, the script prints out the following text, instead of the text printed in the Listing 9.11:

```
Hello Joe
```

The original engine scope is not altered, and its mappings are not changed by the script execution.

ADVANCED EXAMPLE

Consider the following application requirement; we need an easy way to define and change a mathematical formula in the Java application. Let's assume now, for simplicity, that the formula takes only one variable (named `x`). The result should be passed back to the application in a variable named `result`.

We implement this requirement by allowing formulas to be written in an arbitrary scripting language. Then, we use the Scripting API to evaluate those formulas. Along with this example, I demonstrate a few more features related to the engine scope.

Let's say that the `expression.js` script is used for the formula definition and that it looks like the following one.

```
result = 10 * Math.log(Math.pow(x,2));
```

Now look at the Java application shown in Listing 9.13.

Listing 9.13 Advanced Binding Example—Java Application

```
package net.scriptinginjava.ch9;

import java.io.FileReader;

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

public class Expression {
```

Listing 9.13 Continued

```
    public static void main(String[] args) throws Exception {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("js");
        engine.put("x", new Integer(5));
        engine.eval(
            new FileReader(
                "net/scriptinginjava/ch9/expression.js"
            )
        );
        System.out.println(engine.get("result"));
    }
}
```

At first glance, this script looks the same as our previous examples, but a few things are worth explaining in more detail. First, you can see the use of the `put()` and `get()` methods defined in the `ScriptEngine` interface. These methods are just shortcuts for variable binding to the engine scope. So basically, the following statement

```
engine.put("x", new Integer(5));
```

is equal to this:

```
engine.getBindings(ScriptContext.ENGINE_SCOPE)
  .put("x", new Integer(5));
```

And the following

```
engine.get("x")
```

is just a shortcut for this statement:

```
engine.getBindings(ScriptContext.ENGINE_SCOPE).get("x")
```

If this Java program is executed, the value `5` will be passed to the mathematical formula defined in the JavaScript script, and the following result value will be printed:

```
32.18875824868201
```

We can learn one more interesting thing from this example. We can see that the `result` variable was registered in the engine scope from the script, without first being bound from the application. This means every variable initialized in the script will be available for the host application. The state saved in the engine scope is then passed to the next script that is executed with the same engine.

Reserved Keys

Aside from key/value pairs that represent bindings between scripts and application variables, a few keys have special meanings. Table 9.1 lists the reserved keys and their properties, and explains what the values mean.

Table 9.1 Reserved Keys

Key	Property	Meaning of Value
`javax.script.argv`	`javax.script.ScriptEngine.ARGV`	An object array used to pass command-line arguments to the script, where it is appropriate
`javax.script.filename`	`javax.script.ScriptEngine.FILENAME`	The resource or filename of the current script
`javax.script.engine`	`javax.script.ScriptEngine.ENGINE`	The name of the current script engine, as defined by the corresponding `ScriptEngineFactory`
`javax.script.engine_version`	`javax.script.ScriptEngine.ENGINE_VERSION`	The version of the script engine that is used to evaluate the script
`javax.script.language`	`javax.script.ScriptEngine.LANGUAGE`	The name of the language supported by the script engine that is used to execute the script
`javax.script.language_version`	`javax.script.ScriptEngine.LANGUAGE_VERSION`	The scripting language version

Script engine implementation is not required to provide mappings to all of these keys. Also, the engine could define some additional reserved keys that are meaningful to it, in

which case you should consult its documentation. However, you should avoid using keys that start with `javax.script` because they are reserved for usage in future versions of the specification.

In Table 9.1, you can also see that these values are defined in the `ScriptEngine` class as constant fields. Knowing all of this, we can write the Java program shown in Listing 9.14.

Listing 9.14 Reserved Keys

```
package net.scriptinginjava.ch9;

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

public class ParameterTest {

    public static void main(String[] args) {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("js");
        System.out.println(
            engine.getFactory().getLanguageName()
        );
        System.out.println(
            engine.getFactory().getParameter(
                ScriptEngine.LANGUAGE
            )
        );
        System.out.println(engine.get(ScriptEngine.LANGUAGE));
    }
}
```

The first two bolded statements should return the same value. But as I said, the binding in the engine scope is optional, so the third method in this case will return the `null` value.

Global Scope

As already stated, the global scope is related to the `ScriptEngineManager`. All bindings made in it are available to all scripting engines created by that manager.

Let's demonstrate the differences between engine and global scope with a simple example. The following Groovy script just prints two variables on standard output (`global.groovy`):

> **NOTE**
> To run this example, you need to have a Groovy engine in your classpath.

```
println host;
println engine;
```

Here is the equivalent script written in JavaScript (global.js):

```
println(host);
println(engine);
```

Now look at the Java application shown in Listing 9.15.

Listing 9.15 Global Scope Example

```
package net.scriptinginjava.ch9;

import java.io.FileReader;

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

public class GlobalTest {

    public static void main(String[] args) throws Exception {
        ScriptEngineManager manager = new ScriptEngineManager();

        Bindings bindings = manager.getBindings();
        bindings.put("host", "GlobalTest application");

        ScriptEngine engine1 =
            manager.getEngineByName("groovy");
        ScriptEngine engine2 =
            manager.getEngineByName("js");

        engine1.put("engine", "First engine");
        engine1.eval(
            new FileReader(
                "net/scriptinginjava/ch9/global.groovy"
            )
        );

        engine2.put("engine", "Second engine");
        engine2.eval(
            new FileReader(
                "net/scriptinginjava/ch9/global.js"
            )
        );
    }
}
```

In this program, we first initialized a script engine manager, as we did in all of our previous examples. Next, we used the getBindings() method to get the manager's namespace—in other words, the global scope. We set the value for the host variable in this scope. Finally, we initialized two engines,

Groovy and JavaScript, set different values for the `engine` variables in the engine scopes of those two engines, and evaluated the previously defined scripts.

As a result, we have the following text on standard output:

```
GlobalTest application
First engine
GlobalTest application
Second engine
```

We can see that the global scope in each engine is the same. That scope is set in the engine when the manager initializes it through one of the `getEngineBy`*`XXX`*`()` methods. On the other hand, both engines have their own engine scopes, which are entirely independent.

OVERRIDING VARIABLES OF THE GLOBAL SCOPE

If the same value is defined in the global and engine scopes, engine scope binding will override the global scope (see Listing 9.16).

Listing 9.16 Variables Overriding—An Example

```
package net.scriptinginjava.ch9;

import java.io.FileReader;

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

public class OverrideTest {

    public static void main(String[] args) throws Exception {
        ScriptEngineManager manager = new ScriptEngineManager();

        manager.put("host", "GlobalTest application");

        ScriptEngine engine1
            = manager.getEngineByName("groovy");
        ScriptEngine engine2
            = manager.getEngineByName("js");

        engine1.put("engine", "First engine");
        engine1.eval(
            new FileReader(
                "net/scriptinginjava/ch9/global.groovy"
            )
        );
```

Listing 9.16 Continued

```
        engine2.put("engine", "Second engine");
        engine2.put("host", "Overridden value");
        engine2.eval(
            new FileReader(
                "net/scriptinginjava/ch9/global.js"
            )
        );
    }
}
```

In this example, we put the value for the `host` key in the engine scope of the second engine. When executed, the Java application produces the following output:

```
GlobalTest application
First engine
Overridden value
Second engine
```

As you can see, the value of the `host` variable binding is changed.

Shortcut Methods

As was the case with the engine scope and the `ScriptEngine` interface, the `ScriptEngineManager` class provides methods that you can use to put and get values in the global scope (see Listing 9.17).

Listing 9.17 `ScriptEngineManager`'s Shortcut Methods

```
package net.scriptinginjava.ch9;

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

public class CreateGlobalTest {

    public static void main(String[] args) throws Exception {
        ScriptEngineManager manager = new ScriptEngineManager();
        manager.put("name", "Dejan");

        ScriptEngine engine = manager.getEngineByName("js");
        engine.eval("println(name)");

    }
}
```

Listing 9.17 prints out the following on standard output:

```
Dejan
```

In Listing 9.17, the `put()` and `get()` methods are shortcuts to the `put()` and `get()` methods of the global scope.

Global Scope Initialization

An important role of the script engine manager is to initialize the global scope in engines it creates. In all engines created using the script engine manager's `getEngineBy`*`XXX`*`()` methods, their namespace is set as a global scope of the engine.

Of course, you can always instantiate the engine directly through its constructor, but then the global scope is not initialized, and you are responsible for that task too. Take the Java program in Listing 9.18, for example.

Listing 9.18 Global Scope Initialization

```
package net.scriptinginjava.ch9;

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

import com.sun.script.javascript.RhinoScriptEngine;

public class CreateGlobalTest {

    public static void main(String[] args) throws Exception {
        ScriptEngineManager manager = new ScriptEngineManager();
        manager.put("name", "Dejan");

        ScriptEngine engine = new RhinoScriptEngine();
        engine.eval("println(name)");

    }
}
```

In Listing 9.18, we created a manager instance and put mapping in its global scope namespace. After that, we created an instance of the `RhinoScriptEngine` class directly (the bold code). At the end, we evaluated the script that uses the `name` variable expected in the global scope. Because this engine is not instantiated through the manager, its global scope is not initialized, and the program will throw the following exception:

```
sun.org.mozilla.javascript.internal.EcmaError:
  ReferenceError: "name" is not defined.
    (<Unknown source>#1) in <Unknown source>
  at line number 1
```

We could, of course, initialize the global scope through the engine's `setBindings()` method in this case. This approach is well suited if you are going to use only one engine in your project. If the engine should be defined dynamically according to the file extension, language name, or mime type of the resource, you should certainly use the manager's lookup methods.

Script Context

Thus far, we have seen how global and engine scopes differ and what purpose they can have in our script-aware Java applications. These two scopes are an integral part of one more general structure that represents the state of the script engine. This abstraction is defined in the `javax.script.ScriptContext` interface.

Namespaces

`ScriptContext` is the set of namespaces that represent the state of the script engine. These namespaces are also available to the scripts executed in a particular engine.

We already saw this in our examples of global and engine scopes. These two scopes are part of the context defined for the current script engine. We could perform various operations on it, such as modifying it, replacing a certain namespace, or even replacing the whole engine's context. Listing 9.19 demonstrates methods that we could use for operations on the engine's context.

Listing 9.19 Namespace Example

```
package net.scriptinginjava.ch9;

import java.io.FileReader;

import javax.script.SimpleScriptContext;
import javax.script.Bindings;
import javax.script.ScriptContext;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
```

Listing 9.19 Continued

```
import javax.script.SimpleBindings;

public class ContextTest {

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

        Bindings globalNS = new SimpleBindings();
        globalNS.put("host", "ContextTest application");

        Bindings engineNS = new SimpleBindings();
        engineNS.put("engine", "Context demo engine");

        SimpleScriptContext context = new SimpleScriptContext();
        context.setBindings(globalNS
            , ScriptContext.GLOBAL_SCOPE);
        context.setBindings(engineNS
            , ScriptContext.ENGINE_SCOPE);

        engine.setContext(context);
        engine.eval(
            new FileReader("net/scriptinginjava/ch9/global.js")
        );
    }
}
```

The specification defines the `javax.script.SimpleScriptContext` class, which is a default implementation of the `javax.script.ScriptContext` interface. In Listing 9.19, we created an instance of the `SimpleScriptContext` class. Next, we set two namespaces in it: one global scope and one engine scope. At the end, we set the context instance in our engine and evaluated a script. The script we used is the one from the example that shows the difference between global and engine scopes. It is a simple JavaScript code that prints out the values of the `host` and `engine` variables. When executed, this Java program prints the following result on standard output:

```
ContextTest application
Context demo engine
```

This simple example demonstrates that the global and engine scopes we used earlier, directly through methods of the `ScriptEngineManager` and `ScriptEngine` abstractions, are contained in the `ScriptContext` state of the engine. We

also saw that you can manipulate the engine's namespaces through their context, if you find it more appropriate in certain situations.

You can also pass the `ScriptContext` to the `ScriptEngine`'s `eval()` method. By doing this, the context of the engine is overridden with the submitted context. This is demonstrated in Listing 9.20.

Listing 9.20 Evaluating a Script in the Script Context

```
package net.scriptinginjava.ch9;

import java.io.FileReader;

import javax.script.SimpleScriptContext;
import javax.script.Bindings;
import javax.script.ScriptContext;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.SimpleBindings;

public class ExecutionContextTest {

    public static void main(String[] args) throws Exception {
        ScriptEngineManager manager = new ScriptEngineManager();
        manager.put("host", "Test host");
        ScriptEngine engine = manager.getEngineByName("js");
        engine.put("engine", "Test engine");

        Bindings globalNS = new SimpleBindings();
        globalNS.put("host", "ContextTest application");

        Bindings engineNS = new SimpleBindings();
        engineNS.put("engine", "Context demo engine");

        SimpleScriptContext context = new SimpleScriptContext();
        context.setBindings(globalNS
            , ScriptContext.GLOBAL_SCOPE);
        context.setBindings(engineNS
            , ScriptContext.ENGINE_SCOPE);

        System.out.println("- original context -");
        engine.eval(
            new FileReader("net/scriptinginjava/ch9/global.js")
        );
        System.out.println("- modified context -");
        engine.eval(
            new FileReader("net/scriptinginjava/ch9/global.js")
          , context
        );
    }
}
```

Here, we initialized the engine with its context, through the put() methods of the manager and the engine itself. Next, we created a SimpleScriptContext instance as in the previous example. Finally, we executed the same script twice. First we did it using the eval() method, as we did in all our previous examples. This eval() method used the engine's original context. In the second eval() call, we passed the context that we created independently of the engine (and manager). As a result, we can expect the following output from the application:

```
- original context -
Test host
Test engine
- modified context -
ContextTest application
Context demo engine
```

As you can see, the second call used the provided context. It is important to note that this provided context does not replace the original context; it is just used as the state in this particular call. Also, the original engine's context cannot be changed by the script execution, as it can in the case of normal calls.

ATTRIBUTES

The ScriptContext interface and its SimpleScriptContext implementation define some additional methods that we can use for binding tasks both in Java applications and in scripts.

As we already saw, objects with the same name can be defined in more than one namespace (scope). Because the engine context holds all the namespaces, there has to be a way to prioritize them. Namespaces are mapped to int-valued keys, when they are put in the context. We have used two keys thus far:

- ScriptContext.ENGINE_SCOPE, with a value of 100 that represents the namespace defined on the engine's level
- ScriptContext.GLOBAL_SCOPE, with a value of 200 that represents the namespace defined on the manager's level

As we see in a moment, application developers can also define other scopes of interest.

The value of the key is the priority of the scope in the context. The lower the value, the higher priority the scope has.

The `getAttributesScope()` method of the `ScriptContext` interface returns the lowest scope in which an attribute is defined (see Listing 9.21).

Listing 9.21 Determining the Attribute's Scope

```
package net.scriptinginjava.ch9;

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

public class AttributeTest {

     public static void main(String[] args) throws Exception {
          ScriptEngineManager manager = new ScriptEngineManager();

          manager.put("host", "GlobalTest application");

          ScriptEngine engine = manager.getEngineByName("js");

          engine.put("engine", "Second engine");
          engine.put("host", "Overridden value");

          int scope = engine.getContext()
                         .getAttributesScope("host");

          switch (scope) {
               case ScriptContext.GLOBAL_SCOPE :
                    System.out.println("It is in the global scope");
                    break;
               case ScriptContext.ENGINE_SCOPE :
                    System.out.println("It is in the engine scope");
                    break;
               default :
                    System.out.println("Scope unknown");
          }

     }
}
```

In this example, we initialized the engine and its state. Notice that the `host` variable is defined both in global and in engine scopes. So if you run the application, you will get the following message:

```
It is in the engine scope
```

You get this message because the engine scope has a lower priority than the global scope. You can try to experiment and comment the definition of the `host` variable in the engine scope. The result shows that the variable is now defined in the global scope.

The `getAttributesScope()`method could be used by the `Object getAttribute(String name)` method of the `ScriptContext` interface implementation. This `getAttribute()` method returns the value of the attribute defined in the lowest scope (or `null` if the attribute with that name is not defined in any of the context's scopes).

You can also pass the desired scope as the second argument to the `getAttribute()` method to specify the exact namespace you want to use. The `setAttribute()` method signatures matches the appropriate `getAttribute()` method signatures. These methods are demonstrated in Listing 9.22.

Listing 9.22 Obtaining an Attribute from the Desired Scope

```
package net.scriptinginjava.ch9;

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

public class AttributeTest1 {

    public static void main(String[] args) throws Exception {
        ScriptEngineManager manager = new ScriptEngineManager();

        manager.put("host", "GlobalTest application");

        ScriptEngine engine = manager.getEngineByName("js");

        engine.put("engine", "Second engine");
        engine.put("host", "Overridden value");

        System.out.println(
            engine.getContext().getAttribute(
                "host", ScriptContext.GLOBAL_SCOPE
            )
        );
        System.out.println(
            engine.getContext().getAttribute("host")
        );

    }
}
```

The application's execution result is as follows:

```
GlobalTest application
Overridden value
```

This occurs because the first method looks up the `host` variable in the global scope, and the second one gets the value from the engine scope because it has a lower key value.

The global context also is important for scripts that are evaluated. Besides variable values mapped to script variables, the `context` script variable represents the global context and can be used for its manipulation (see Listing 9.23).

Listing 9.23 Modifying Attributes in `script`—Script

```
context.setAttribute("engineKey"
                    , "engine scope variable, set in script"
                    , 100
);
globalKey = "global scope variable, changed in script";
```

The script in Listing 9.23 sets the value for the `engineKey` attribute in the engine scope (namespace key value 100) and the `globalKey` in the global scope (by changing the value of the script variable). Consider now the Java program shown in Listing 9.24, which evaluates this script.

Listing 9.24 Modifying Attributes in `script`—Java Application

```
package net.scriptinginjava.ch9;

import java.io.FileReader;

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


public class GlobalPutTest {

     public static void main(String[] args) throws Exception {
          ScriptEngineManager manager = new ScriptEngineManager();
          manager.put(
               "globalKey"
             , "global scope, set in application"
          );
          ScriptEngine engine = manager.getEngineByName("groovy");
          System.out.println(manager.get("globalKey"));
          engine.eval(
               new FileReader("net/scriptinginjava/ch9/put.groovy")
          );
```

Listing 9.24 Continued

```
        System.out.println(manager.get("globalKey"));
        System.out.println(engine.get("engineKey"));
    }
}
```

The application prints the following output:

```
global scope variable, set in application
global scope variable, changed in script
engine scope variable, set in script
```

The first line is printed from the Java application before the script has been evaluated. The value printed is the value of the `globalKey` attribute set by the application itself. Next, the script is evaluated and the script changes the value of the global `globalKey` attribute. The value set by the script is printed by the application after the evaluation. Finally, the value of the `engineKey` attribute set in the script is printed.

Although you will probably use only the `get()` and `put()` methods of the manager and engine objects in most cases, it is good to be aware of these additional capabilities of the Scripting API for variable value binding.

Custom Namespaces

As I explained earlier in this chapter, the script context is a set of namespaces (scopes). Thus far, I have described techniques for dealing with just two of them: engine scope and global scope. In some applications, you need to create custom scopes for data binding. For example, in the following chapter, we discuss the Web Scripting Framework (the `javax.script.http` package) that is created using this Scripting API.

Web applications have three more scopes of interest: the request scope, session scope, and application scope. In this section, I cover the steps you need to take to create contexts with custom scopes for your project.

A default implementation of the `ScriptContext` interface is the `javax.script.SimpleScriptContext` class. This class is used by default in all script engine implementations.

The only issue with this class is that it is capable only of mapping namespaces with key values that represent engine and global scopes (values 100 and 200, respectively). This is expected, because the Scripting API's specification defines just these two scopes as required. So if you try to add a namespace in `SimpleScriptContext` with a key that does not have one of these two specified values, you'll get a `java.lang.IllegalArgumentException` exception.

This behavior is demonstrated in Listing 9.25.

Listing 9.25 Defining a Custom Namespace

```
package net.scriptinginjava.ch9;

import javax.script.Bindings;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.SimpleBindings;

public class CustomTest {

    public static void main(String[] args) {
        ScriptEngineManager manager = new ScriptEngineManager();

        Bindings customNS = new SimpleBindings();

        ScriptEngine engine = manager.getEngineByName("js");
        ScriptContext context = new SimpleScriptContext();
        context.setBindings(customNS, 1000);
    }
}
```

Here, we created a new `SimpleScriptContext` class instance and tried to set the namespace with the key value 1000. As a result, the application throws the following exception:

```
java.lang.IllegalArgumentException: Invalid scope value.
```

To create and use custom scopes, we have to create a specialized context interface implementation. You can find one example of such an implementation in Listing 9.26.

Listing 9.26 Custom Application Context

```
package net.scriptinginjava.ch9;

import javax.script.SimpleScriptContext;
import javax.script.Bindings;
```

Listing 9.26 Continued

```
public class ApplicationContext extends SimpleScriptContext {

    public static final int APPLICATION_SCOPE = 1000;

    protected Bindings applicationScope;

    public void setBindings(Bindings bindings, int scope) {
        if (scope == APPLICATION_SCOPE)
            applicationScope = bindings;
        else
            super.setBindings(bindings, scope);
    }

    public Bindings getBindings(int scope) {
        if (scope == APPLICATION_SCOPE)
            return applicationScope;
        else
            return super.getBindings(scope);
    }

    public void setAttribute(java.lang.String name
        , java.lang.Object value, int scope) {
        if (scope == APPLICATION_SCOPE)
            applicationScope.put(name, value);
        else
            super.setAttribute(name, value, scope);
    }

    public Object getAttribute(String name, int scope) {
        if (scope == APPLICATION_SCOPE)
            return applicationScope.get(name);
        else
            return super.getAttribute(name, scope);
    }

    public Object getAttribute(String name) {
        Object retVal = super.getAttribute(name);
        if (retVal != null)
            return retVal;
        else
            return applicationScope.get(name);
    }

    public int getAttributesScope(String name) {
        int scope = super.getAttributesScope(name);
        if (scope != -1)
            return scope;
        else {
            if (applicationScope.get(name) != null)
                return APPLICATION_SCOPE;
            else
                return -1;
        }
    }

    public Object removeAttribute(String name, int scope) {
        if (scope == APPLICATION_SCOPE)
            return applicationScope.remove(name);
```

Listing 9.26 Continued

```
        else
            return super.removeAttribute(name, scope);
    }
}
```

Here, we defined an `ApplicationContext` class that extends a default `SimpleScriptContext` implementation. We have overridden all methods that have something to do with setting and obtaining attributes and thus allowed additional scope (with the key value 1000) to be used in the application.

Now we can write a Java application that uses this new script context implementation and the appropriate script (see Listing 9.27).

Listing 9.27 Using Custom Application Context

```
package net.scriptinginjava.ch9;

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

public class NewCustomTest {

    public static void main(String[] args) {
        ScriptEngineManager manager = new ScriptEngineManager();
        manager.put("global", "global value");

        Bindings customNS = new SimpleBindings();
        customNS.put("key", "value");

        ScriptEngine engine = manager.getEngineByName("groovy");
        ScriptContext context = new ApplicationContext();
        context.setBindings(customNS
            , ApplicationContext.APPLICATION_SCOPE
        );
        context.setBindings(
            engine.getBindings(ScriptContext.GLOBAL_SCOPE)
          , ScriptContext.GLOBAL_SCOPE
        );
        engine.setContext(context);
        engine.eval(
            new FileReader(
                "net/scriptinginjava/ch9/global.groovy"
            )
        );

}
```

Here, we have an `ApplicationContext` instance, and we initialized it with the manager's global scope. We also created a `SimpleBindings` object and set it to be an application scope of our context. Finally, we set it to be an engine's context and evaluated the Groovy script. The Groovy script that demonstrates this context could look like this:

```
println “””${context.getAttribute(“key”)} -\
 ${context.getAttributesScope(“key”)}”””
println “””${context.getAttribute(“global”)} -\
 ${context.getAttributesScope(“global”)}”””
```

The script just prints out the values for the `key` and `global` attributes, and the scopes in which they are found. As a result, we have the following output:

```
value - 1000
global value - 200
```

This output indicates that the `key` attribute is in our custom application scope, and the `global` attribute is located in the engine's global scope.

READERS AND WRITERS

In addition to taking care of the namespaces available to the script engine, the script context also manages reader and writer objects that evaluated scripts use for input and output operations.

For example, say you want to redirect the standard output of your script to some file on your system. You can do it by using the `setWriter()` method of the `ScriptContext` instance of your engine, as shown in Listing 9.28.

Listing 9.28 Custom Engine Writer Example

```
package net.scriptinginjava.ch9;

import java.io.FileReader;
import java.io.FileWriter;

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

Listing 9.28 Continued

```
public class ContextWriterTest {

    public static void main(String[] args) throws Exception {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("groovy");
        engine.getContext().setWriter(
            new FileWriter("C:\\out.log")
        );
        engine.eval(
            new FileReader(
                "net/scriptinginjava/ch9/write.groovy"
            )
        );
        engine.getContext().getWriter().close();
    }
}
```

In this example, we redirected the output of the evaluated script to the file named `C:\out.log`. If we evaluate the following script with the program in Listing 9.28, you can expect that the text printed with the `println` Groovy command will be written in the specified file:

```
println("""This text should be written to the file \
, not console""");
```

In the same manner, you can redirect the script's standard input and standard error streams using the `setReader()` and `setErrorWriter()` methods, respectively. Of course, you could use other resources with their reader and writer objects, instead of files.

Code Generation

You are meant to use the Scripting API as a general scripting framework that enables a unique interface to various engines. Thus, it provides some basic script generation methods. I mentioned earlier that the `ScriptEngineFactory` interface provides basic metadata that describes certain engine implementations. Besides that interface, three additional mechanisms—`getOutputStatement()`, `getMethodSyntaxCall()`, and `getProgram()`—could help application developers to create basic script constructs that are usually found in programming languages. In this section, I walk you through these mechanisms.

Output Statement

Probably the most frequently used statements in scripting languages are those that print values to standard output. Even though their names vary—some are called `print()`, some `echo()`, and so on—their purpose and syntax are almost identical.

The `getOutputStatement()` method of the `ScriptEngineFactory` interface is used to format the string passed as an argument to the statement that prints it in a given scripting language. Let's demonstrate this method with a simple example, shown in Listing 9.29.

Listing 9.29 Output Statement

```
package net.scriptinginjava.ch9;

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


public class OutputTest {

    public static void main(String[] args) {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("js");
        String testValue = "Hello world";
        String statement =
            engine.getFactory().getOutputStatement(testValue);
        System.out.println(statement);
    }
}
```

The value of the `statement` variable after the call to the `getOutputStatement()` method is as follows:

```
print("Hello world")
```

In this example, we just printed this value for demonstration purposes, but as we see later in this section, you can evaluate it as a part of a larger script.

Method Call Syntax

Another often-used operation is execution of the objects' methods. The Scripting API provides the `getMethodCallSyntax()`

method, defined in the `ScriptEngineFactory` interface, which generates a statement that you can use to invoke the method of a Java object.

In this case, the syntax of these calls could vary a great deal among languages, even though the basic principle is the same. For example, variables in PHP start with a `$`, and instead of a `.` character for method invocation, the `->` operator is used.

So, the following syntax in PHP:

```
$obj->method($arg1, $arg2)
```

is the same as this syntax in JavaScript:

```
obj.method(arg1, arg2)
```

Listing 9.30 demonstrates how we can use the Scripting API to create a JavaScript method call from this example.

Listing 9.30 Method Call Syntax

```
package net.scriptinginjava.ch9;

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


public class MethodSyntaxTest {

    public static void main(String[] args) {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("js");
        String statement =
            engine.getFactory().getMethodCallSyntax(
                "obj", "method", new String[] {"arg1", "arg2"}
            );
        System.out.println(statement);
    }
}
```

In Listing 9.30, we demonstrated the syntax of the `getMethodCallSyntax()` method. The first argument is the name of the object (variable) whose method we want to invoke. The second argument represents the name of the method, and the last one is an array of strings representing arguments that should be passed to the method call.

The value of the `statement` variable after the `getMethodCallSyntax()` method is called is the

```
obj.method(arg1,arg2)
```

In this example, we just printed it to standard output, but a statement like this one would usually be used to create more complex scripts.

Program

Besides generating single statements, the Scripting API provides the `String getProgram(String...)` method, which is used to arrange into a script a series of statements given in the form of an array of `String` arguments.

PHP scripts, for example, consist of statements that are delineated by semicolons and enclosed in a `<? . . . ?>` block, while JavaScript scripts are just statements delineated by semicolons as shown in Listing 9.31.

Listing 9.31 Program Example

```
package net.scriptinginjava.ch9;

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

public class ProgramTest {

    public static void main(String[] args) {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("js");
        ScriptEngineFactory factory = engine.getFactory();
        String methodCall =
            factory.getMethodCallSyntax(
                "scriptObject", "scriptMethod"
              , new String[] {"arg1", "arg2"}
            );
        String output = factory.getOutputStatement("Hello");
        String script =
            factory.getProgram(
                new String[] {methodCall, output}
            );
        System.out.println(script);
    }
}
```

In Listing 9.31, we used all three of the "code generation" methods covered in this section. First, we created a method object call that produced the `scriptObject.scriptMethod(arg1,arg2)` statement. Next, we created an output statement, `print("Hello")`, and then those two statements were separated with the `;` char. As a result, we get the following script:

```
scriptObject.scriptMethod(arg1,arg2);print("Hello");
```

You can evaluate the script using the `eval()` method, for example, or save it in a file for later use.

Additional Engine Interfaces

Thus far, we have seen how we can use the Scripting API to evaluate scripts and how the data binding mechanism works in this library. The general idea used in this API is to have most of the basic functionalities that the scripting engine should implement defined in the `ScriptEngine` interface. Every additional requirement, such as invocation of functions or compilation of scripts to Java bytecode, is defined in a separate interface. This design approach guarantees a minimum number of changes to these interfaces in the future and ensures that engines with minimal functionalities comply with the specification.

In this section, I cover two additional interfaces that scripting engines could implement to provide more functionality to users:

- **`javax.script.Invocable`**—Implemented by engines that can invoke functions defined in scripts
- **`javax.script.Compilable`**—Implemented by engines that can compile scripts to bytecode

Invocable

In our discussion of the BSF API, we saw that another important task of scripting frameworks is the invocation of functions defined in scripts. For that purpose, the Scripting API defines a `javax.script.Invocable` interface.

FUNCTIONS

Let's start exploring the Invocable interface, through concrete examples. Recall the example shown in Listing 9.13, in which we defined a mathematical formula in a script and evaluated it with the Java application. Now, we are going to expand on this example. To begin, we are going to define the formula as the script function (procedure):

```
def expression(x) {
    return 10 * Math.log(Math.pow(x,2));
}
```

The function is called expression, and it takes one argument, named x. The return value is, of course, the value of the expression for the given argument. Now let's use the Scripting API to evaluate expressions using this function (see Listing 9.32).

Listing 9.32 Invocable Example

```
package net.scriptinginjava.ch9;

import java.io.FileReader;

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

public class FunctionTest {

     public static void main(String[] args) throws Exception {
          ScriptEngineManager manager = new ScriptEngineManager();
          ScriptEngine engine = manager.getEngineByName("groovy");
          if (engine instanceof Invocable) {
               engine.eval(
                    new FileReader(
                         "net/scriptinginjava/ch9/function.groovy"
                    )
               );
               Object result =
                    ((Invocable)engine).invokeFunction(
                         "expression", new Object[]{new Integer(5)}
                    );
               System.out.println(result);
          } else {
               System.out.println(
                    "The engine is not able to invoke functions"
               );
          }
     }
}
```

The first important thing we have to do is to ensure that the engine implementation we are going to use can invoke function calls. To do this, we need to know whether the engine is an instance of the `Invocable` interface because this guarantees such capability.

If this condition is met, we have to evaluate the script using the `eval()` method of the `ScriptEngine` interface, and finally, invoke the function using the `invokeFunction()` method of the `Invocable` interface.

The `invokeFunction()` method accepts two arguments. The first argument is the name of the function that should be invoked, and the second one is an array of objects that represents the parameter list to be passed to the function. In this case, we provided just one integer argument with a value of `5`. The `invokeFunction()` method returns an object that represents the return value of the function. This example prints the same result as the original demo that only evaluated this expression.

Methods

You also could use the `Invocable` interface to invoke methods on the objects defined in the script. Let's continue to work on the mathematical formula example, by converting our function into a method of a class named `MathUtil`:

```
class MathUtil {

    def expression(x) {
        return 10 * Math.log(Math.pow(x,2));
    }

}

mathUtil = new MathUtil();
```

In this Groovy script, we defined the `MathUtil` class and created an instance of it named `mathUtil`. Now we can use the `invokeMethod()` method to invoke the `expression` method of the `mathUtil` object (see Listing 9.33).

Listing 9.33 Method Call

```
package net.scriptinginjava.ch9;

import java.io.FileReader;

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

public class MethodTest {

    public static void main(String[] args) throws Exception {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("groovy");
        if (engine instanceof Invocable) {
            engine.eval(
                new FileReader(
                    "net/scriptinginjava/ch9/method.groovy"
                )
            );
            Object result =
                ((Invocable)engine).invokeMethod(
                    engine.get("mathUtil")
                  , "expression"
                  , new Object[]{new Integer(5)}
                );
            System.out.println(result);
        } else {
            System.out.println(
                "The engine is not able to invoke methods"
            );
        }
    }
}
```

The only difference between the `invokeFunction()` and `invokeMethod()` methods is that the later method has one more argument. It is the script object whose method we want to be invoked. This object is defined in the engine's scope, so we used the `get()` method to get the reference to it.

INTERFACES

As we have seen throughout this book, another important role of scripting languages is the implementation of Java interfaces. The Scripting API defines the `getInterface()` method in the `Invocable` interface for this purpose.

To demonstrate this behavior, let's start by defining a Java interface that holds an `expression` method definition (the one that will be implemented in the script):

```
package net.scriptinginjava.ch9;

public interface MathUtil {

    public Double expression(Integer x);

}
```

Because the Scripting API enables you to implement Java interfaces by implementing their methods as the script's functions and procedures, we can use the script that we already wrote for the function call example. Just to recall, that script could look like this:

```
function expression(x) {
    return 10 * log(pow(x,2));
}
```

The signature of this Groovy function matches the signature of the expression method of the `MathUtil` interface, and because it is the only method defined in that interface, we can say that this script complies with the interface. Now let's instantiate it as a Java object (see Listing 9.34).

Listing 9.34 Interfaces

```
package net.scriptinginjava.ch9;

import java.io.FileReader;

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

public class InterfaceTest {

    public static void main(String[] args) throws Exception {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("groovy");
        if (engine instanceof Invocable) {
            engine.eval(
                new FileReader(
                    "net/scriptinginjava/ch9/function.groovy"
                )
            );
            MathUtil math =
                ((Invocable)engine).getInterface(MathUtil.class);
            System.out.println(math.expression(new Integer(5)));
        } else {
            System.out.println(
```

Listing 9.34 Continued

```
                "The engine is not able to invoke functions"
            );
        }
    }
}
```

The application takes all the standard steps that we already saw in this chapter's examples. After we checked that the engine implements the desired interface and evaluated the script, we called the `getInterface()` method. This method takes an interface class as an argument and returns an object that complies with the interface. After this call, the returned objects can be used as regular Java objects.

Compilable

One of the areas where the BSF did not meet expectations is support for compilation of scripts to the intermediate code. As I mentioned in the introductory chapters of this book, many script languages internally compile scripts to some intermediate language that is faster to interpret than the original source. Some of them are capable of storing and re-executing the intermediate code.

The Scripting API provides a standard solution to this task for engine implementations that need it. Just as was the case with invocation of functions and methods, the API defines a separate interface that needs to be implemented by engines with this functionality. This interface is `javax.script.Compilable`, and I demonstrate it by an example.

To begin, let's define a simple Groovy script:

```
println "Hello"
```

I chose Groovy because it supports script compilation to the Java bytecode. Practically all Java-enabled scripting languages provide this functionality.

Now let's see how we can compile and execute this Groovy script and what benefits this approach introduces (see Listing 9.35).

Listing 9.35 Compilable

```
package net.scriptinginjava.ch9;

import java.io.FileReader;

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

public class CompilableTest {

    public static void main(String[] args) throws Exception {

        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("groovy");
        FileReader source =
            new FileReader(
                "net/scriptinginjava/ch9/compile.groovy"
            );


        if (engine instanceof Compilable) {
            CompiledScript bin =
                ((Compilable)engine).compile(source);
            long start = System.currentTimeMillis();
            bin.eval();
            System.out.println(
                "Script execution time: "
              + (System.currentTimeMillis() - start) + " ms"
            );
        }


    }
}
```

First, we have to check whether the current script engine supports this functionality. This is not necessary if we are going to use only one (or a few) script engines that will be statically bound, but it is necessary if we are planning to use the manager's discovery mechanism to prevent a casting exception in our application.

The `Compilable` interface defines the `compile()` method with two signatures. The signature you will use depends on whether you are going to provide the script's source as a `String` or `Reader` object. In this example, we provided the `FileReader` object instance that points to the file containing the script's source.

The `compile()` method returns an instance of the `javax.script.CompiledScript` class. Extensions of this class store the script's intermediate code and provide methods for its execution. The `CompiledScript` class works with the Java class format as intermediate code, but its extensions that are implemented by different script engines could also handle Java class files or a language-specific intermediate code format.

Regardless of which intermediate format is used, the `eval()` method, defined by the `CompiledScript` class and its extensions, is used to execute that code.

For demonstration purposes, I put checkpoints in the code so that we can measure how long it takes for such a compiled script to execute. The following output shows one possible result:

```
Hello
Script execution time: 160 ms
```

To compare the execution of compiled scripts with that of regular scripts, you can run the following Java program:

```
package net.scriptinginjava.ch9;

import java.io.FileReader;

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

public class Compilable1Test {

    public static void main(String[] args) throws Exception {

        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("groovy");
        FileReader source =
            new FileReader(
                "net/scriptinginjava/ch9/compile.groovy"
            );

        long start = System.currentTimeMillis();
        engine.eval(source);
        System.out.println(
            "Script execution time: "
          + (System.currentTimeMillis() - start) + " ms");

    }
}
```

The matching result now looks like this:

```
Hello
Script execution time: 1793 ms
```

It is obvious that for the Groovy scripting language, this difference is relevant, even for the simplest scripts. If performance is an issue in your project, you can try to use the `Compilable` interface and implement the mechanism that will recompile the script every time it changes.

The last issue related to compiled scripts is how to provide the context for compiled scripts such as this one. First, compiled scripts are executed in the instance of the engine that compiled them. You can obtain a reference to that engine by using the `getEngine()` method defined in the `CompiledScript` class.

On top of that, the `eval()` method defined in this class accepts the `Bindings` or `ScriptContext` objects as arguments. By providing these arguments, we can modify the context in which the (compiled, in this case) script will be executed. These mechanisms are the same as the ones we used in the standard script evaluation.

Threading

Earlier in this chapter, we discussed engine metadata and how you can use `ScriptEngineFactory` interface implementations to obtain that information for a particular engine. There is one property that I didn't mention there because it deserves special treatment.

In the Java environment, it is important to know whether a certain object is thread safe—in other words, whether it can be used safely in a multithreaded environment. The same is true for the `ScriptEngine` interface implementations. The important information for application developers is whether they can evaluate scripts concurrently on multiple threads, and what happens with the engine's context in that case.

This information is defined with the `THREADING` parameter of the engine's corresponding `ScriptEngineFactory` object.

The application defines the following meanings for the return values.

- **null**—The engine implementation is not thread safe and cannot be used to execute scripts concurrently on multiple threads.
- **MULTITHREADED**—The engine is thread safe, and it can be used in a multithreaded environment. The only catch is that the evaluation of one script may influence scripts in other threads. This means changes to the symbols' values affect other scripts that use those symbols.
- **THREAD-ISOLATED**—The engine satisfies the requirements of MULTITHREADED engines but guarantees that symbol value changes will not affect scripts executing in other threads.
- **STATELESS**—The engine satisfies the requirements of the THREAD-ISOLATED engines. On top of that, script execution does not alter the engine's scope.

Let's now see how engines found in the JDK behave in a multithreaded environment (see Listing 9.36).

Listing 9.36 Threading

```
package net.scriptinginjava.ch9;

import java.util.List;

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

public class ThreadTest {

    public static void main(String[] args) {
        ScriptEngineManager manager = new ScriptEngineManager();
        List<ScriptEngineFactory> factories =
            manager.getEngineFactories();
        for (ScriptEngineFactory factory: factories) {
            System.out.println(
                factory.getLanguageName()
              + " - "
              + factory.getParameter("THREADING")
            );
        }
    }
}
```

Listing 9.36 loops through all the engines found by the discovery mechanism and prints their language name and threading capabilities. For engines included in Java SE 6, the program would print the following result:

```
ECMA Script - MULTITHREADED
```

As you can see, the Rhino engine is thread safe.

Dynamic Bindings

Earlier in this chapter, we saw how the data-binding mechanism works and how Java objects can be used in scripts. That binding, when the engine is embedded in the host Java application, is called *programmatic binding*.

Another sort of binding is needed for full integration of Java and scripting languages. It is called *dynamic binding*, and it represents the ability of scripting languages to use Java classes and objects when they are not executed through the appropriate `ScriptEngine` interface implementation. Groovy and Rhino naturally support this functionality because their interpreters are implemented in Java. We discussed principles that are used in these languages in Chapters 3 and 4. Thus, we focus on PHP here and discuss how we can integrate native PHP applications with Java.

To be able to use Java from PHP scripts, we have to configure PHP properly. I do not cover the installation and configuration details of PHP here. If you are not familiar with those subjects, consult the PHP manual (www.php.net/manual/en/) before proceeding further with this section.

An example configuration in the `php.ini` file (PHP configuration file) for the Java support could look like Listing 9.37.

Listing 9.37 Example `php.ini` Configuration

```
 [java]
java.home=C:\java\jdk1.5.0_01\jre
java.library.path=C:\dev\jsr223\php5\lib\php
java.share_php_session=0
java.debug_print=0
java.class.path="C:\dev\jsr223\php5\lib\php\script.jar"
java.library=C:\java\jdk1.5.0_01\jre\bin\client\jvm.dll
```

Make sure that you put a configuration similar to this one in the appropriate `php.ini` file on your system. As you can see, through these few initialization properties, you set the location of the Java runtime (`java.home`) and classpath (`java.class.path`) that will be used to look up the Java classes.

Now we are ready to write a PHP script that uses the Java runtime and custom Java libraries. Let's start with a simple example (see Listing 9.38).

Listing 9.38 Dynamic Binding—Java Objects

```
<?

    $javadate = new Java("java.util.Date");
    $date = $javadate->toString();
    echo($date);

?>
```

As you can see, Java classes are created using the `Java` PHP built-in class. The constructor of this class takes the full name of the class as its first argument. Parameters that have to be passed to the constructor (if any) are passed as arguments to the `Java` constructor call. After that object is created, you can use it in standard PHP fashion. If you run this script, in a properly configured environment, by typing `$ php java.php`, on standard output, you get a result similar to the following:

```
Content-type: text/html
X-Powered-By: PHP/5.0.1

Fri Apr 29 12:31:49 CEST 2005
```

In this simple example, we created a `java.util.Date` class instance and invoked the `toString()` method.

Similarly, we can work with Java classes and static methods. Look at Listing 9.39.

Listing 9.39 Dynamic Binding—Java Classes

```
<?
    $thread_class = new JavaClass("java.lang.Thread");
    $thread_object = $thread_class->currentThread();
```

Listing 9.39 Continued

```
    echo "Start: " . date('s') . "\n";
    $thread_class->sleep(1000);
    echo "Checkpoint: " . date('s') . "\n";
    $thread_object->sleep(1000);
    echo "End: " . date('s') . "\n";
?>
```

Here, an instance of the Java class is created through the `JavaClass` constructor call. On this instance, we can make calls to the class's static methods. In this example, we called the `currentThread()` static method and obtained an instance of the `java.lang.Thread` class. Next, we invoked the `sleep()` method to pause execution of the current thread. Note that we did this twice. Once we called it as a static method of the `Thread` class. The second call was on an object as a regular method call. Both calls are regular. We added the code that prints the current second between these calls, so if you run this example by typing `$ php thread.php`, you can expect a result similar to the following to show up on your screen:

```
Content-type: text/html
X-Powered-By: PHP/5.0.1

Start: 04
Checkpoint: 05
End: 06
```

Dynamic bindings are as important as programmatic bindings for many real-life development problems. For scripting languages whose interpreters are implemented in Java, this is sort of a natural thing to do. On the other hand, for native scripting languages such as PHP, this could be tricky to achieve. Engine developers should take care of many issues, such as member selection, argument conversion, and overloaded method resolution. These issues are beyond the scope of this book, and you are advised to read the Scripting API specification for more details on this topic.

Conclusion

The Scripting API is definitely an evolutionary step in scripting framework design. This was expected because it is based on experience taken from APIs such as the BSF and native integration mechanisms provided by languages such as Groovy and BeanShell.

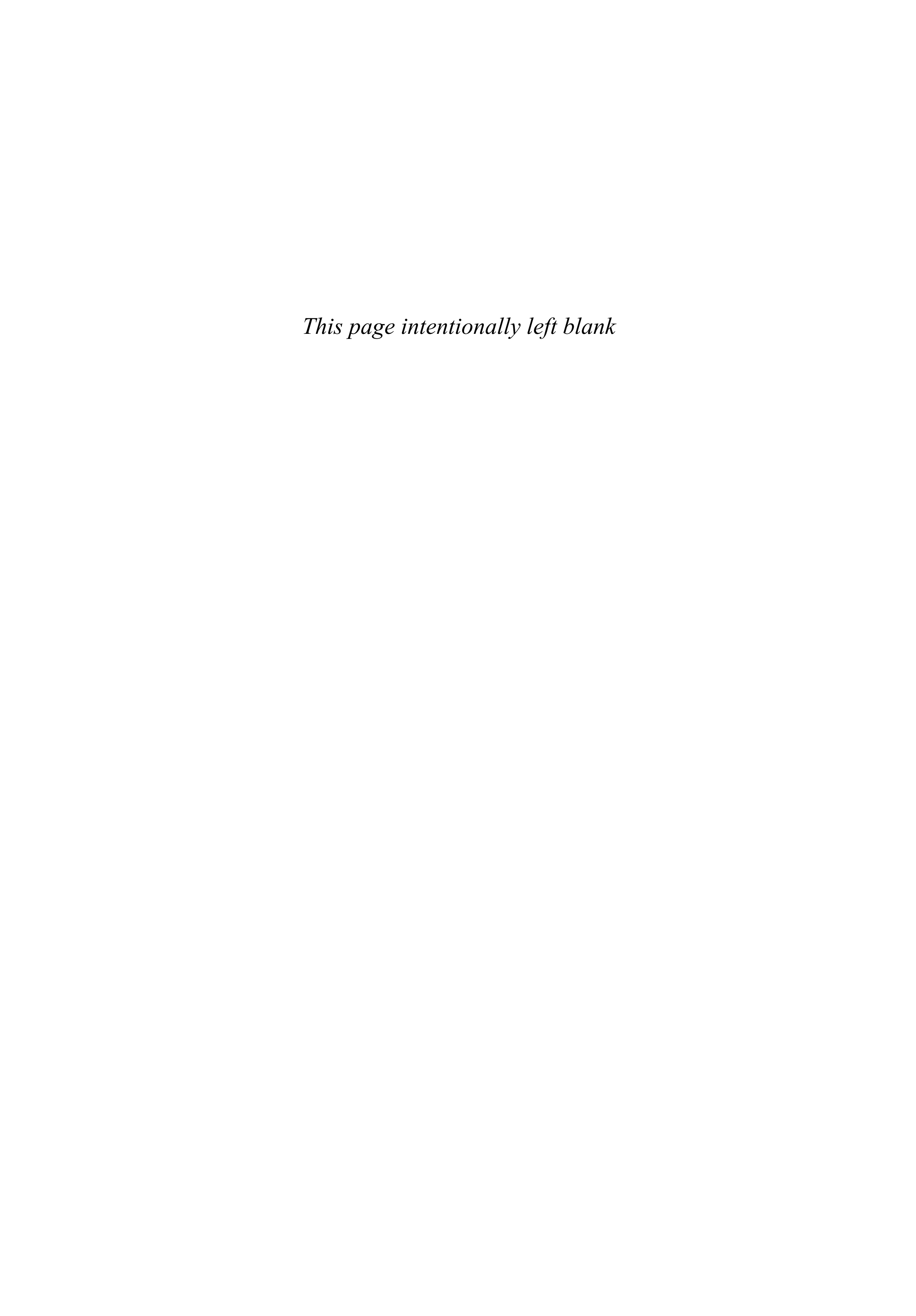
Table 9.2 summarizes the differences between the BSF and Scripting API that I described throughout this chapter.

Table 9.2 Differences Between BSF and Scripting API

Feature	BSF	Scripting API
Discovery mechanism	Static, using the `Languages.properties` file located in the distribution	Dynamic, using the JAR service mechanism that enables a flexible way to register new engine implementations
Evaluation	Two methods, differing in terms of whether the script returns a result	A single method, leading to a cleaner API
Binding	Plain, no support for separating variables according to their purpose	Layered into scopes
Work with script files	Enabled through utility classes	Defined in the engine's interface
Method invocation and script compilation	Implemented in the engine's interface	Implemented in optional interfaces, resulting in a better separation of functionalities

As we discussed in this chapter, the Scripting API has a clean design and separation of concerns, which guarantees a moderate learning curve for developers with experience in similar libraries and for those who are new in this field of development. This API also addressed a few issues related to threading that make it more suitable for implementation of support for native script languages (those that do not have an interpreter implemented in Java).

All these features, and the fact that the Scripting API is included in the Java 6 Standard Edition release onward, make this API a perfect choice as your general scripting framework in Java.


This page intentionally left blank

Chapter 10

Web Scripting Framework

As we already discussed, the initial purpose of the Java Specification Request 223 (JSR 223) was to enable native scripting languages (with PHP as a reference) to generate Web content inside servlet containers. For that purpose, as we have seen, the Scripting API was created as a general framework for integration of scripting interpreters (both native and Java interpreters) and Java applications.

On top of the Scripting API, the Web Scripting Framework was created to address this initial intention. This framework is packaged in the `javax.script.http` package. Although it is removed from the final version of the specification, it represents a valuable source of ideas on which future work will be based. To run examples presented in this chapter, you need to download and install a reference implementation of the specification as described in Appendix C, "Installing JSR 223."

In this chapter, we discuss the basic abstractions in the Web Scripting Framework. I also explain how to configure your Java Web application so that part of it can be implemented in a language such as PHP. You can also use the techniques I present here to integrate Java and PHP Web applications. We discuss that topic in more detail throughout the chapter.

Architecture

To begin, let's see what interfaces and classes comprise this framework. After this brief review of abstractions, it will be much easier to discuss how we can use them.

Context

In Chapter 9, "Scripting API," we discussed the `javax.script.ScriptContext` interface and its default `javax.script.SimpleScriptContext` implementation. We also saw how we can extend these abstractions to adapt them to certain environments.

The `javax.script.http.HttpScriptContext` interface is an extension of the `ScriptContext` abstraction, suited to the Web environment. The `ScriptContext` abstraction represents a set of namespaces, with two basic namespaces defined: engine and global. In the Web environment, however, scripting engines need a few more scopes. Scripts executed in the Web environment deal with attributes defined in the following three additional scopes:

- **Request scope**—This holds attributes mapped to a certain HTTP request. Every URL request by the client's browser is essentially one HTTP request. All attributes passed with that request, using, for example, `GET` or `POST` methods, must be available to the scripting engine. Those variables are available in the request scope defined by the `public static final int REQUEST_SCOPE = 0;` key.

  Later in this chapter, we see how we can use these variables in scripts, and I explain in more detail a mechanism behind these bindings.
- **Session scope**—This holds attributes mapped to a certain client's session. Because the HTTP protocol is stateless, Web applications preserve their states by using sessions bound to a certain user. This data resides on the server side, inside the servlet container. We will not delve into the session mechanism now, and you are advised to

find more information about it in the appropriate literature. For our purposes here, it is important to know that attributes defined in the client's session are available to the scripting engine (and to scripts evaluated with it) in the `HttpScriptContext`'s session scope. This scope is defined with the `public static final int SESSION_SCOPE = 150;` key.

- **Application scope**—Besides requests and sessions that define the Web application context in terms of the user's data, scripting engines need to have access to the servlet container environment. The application scope holds attributes that define the properties of a particular servlet container. For example, scripts can have different code paths depending on whether the underlying container supports this feature. This scope is defined by the `public static final int APPLICATION_SCOPE = 175;` key.

As you can see, first an attribute is looked up in the request scope. Next, the session scope is searched, and finally the lookup is done in the application scope. Although these three scopes are mandatory, some implementations can add more scopes.

Besides this task of providing additional scopes to scripting engines, this interface also represents a bridge between scripting languages and a servlet container. Thus, it must provide a mechanism for configuring a framework's behavior in a certain container or Web application. We see how this mechanism works in the coming sections.

Servlet

Another key abstraction of this framework is the `javax.script.http.HttpScriptServlet` abstract class. You should map this servlet to handle URLs related to scripts, such as URLs with the `.php` extension.

This abstract servlet uses a supplied scripting engine to execute a script in the provided HTTP script context. Therefore, it defines abstract methods used to manipulate these abstractions.

CONTEXT

The `HttpScriptServlet` class extensions should implement the following method:

```
public abstract HttpScriptContext getContext(
    HttpServletRequest request, HttpServletResponse response
) throws ServletException;
```

This method is used to create the context and make the bindings defined earlier. As you can see, the context is created for every client's request because a request scope needs to be initialized. This initialization is done through the following method of the `HttpScriptContext` interface implementations:

```
public void initialize(
    Servlet servlet, HttpServletRequest request
  , HttpServletResponse response
) throws ServletException;
```

SCOPES

Context is initialized with the request and response objects that abstract the HTTP requests and responses. Knowing this, we can conclude that a call to the following method of the context object:

```
getAttribute(key, REQUEST_SCOPE)
```

returns the same value as the following method for every key:

```
request.getAttribute(key)
```

Appropriately, the following method:

```
getAttribute(key, SESSION_SCOPE)
```

returns the same result as this one does:

```
request.getSession().getAttribute(key)
```

And of course, this method:

```
getAttribute(key, APPLICATION_SCOPE)
```

returns the value of the following method call:

```
servlet.getServletConfig().getServletContext()
  .getAttribute(key)
```

ENGINES

Extensions of the `HttpScriptServlet` class must also hold instances of script engines that will be used to process requests. These engines are exposed to the context by the `getEngine()` and `releaseEngine()` methods. The `getEngine()` method has the following signature:

```
public abstract ScriptEngine getEngine(
    HttpServletRequest request
);
```

It is used to look up the appropriate engine to handle a request. It performs this lookup according to the mapping in the appropriate configuration file, as we will see in a moment.

The `releaseEngine()` method is called to indicate that a certain engine is no longer in use. This is important for threading issues, which we cover later in this chapter. The `releaseEngine()` method has the following signature:

```
public abstract void releaseEngine(ScriptEngine engine);
```

Interaction

The reference implementation of the Web Scripting Framework contains default implementations of the `HttpScriptContext` and `HttpScriptServlet` abstractions. So, the `javax.script.http.GenericHttpScriptContext` context is used along with the `com.sun.script.http.ScriptServlet` servlet.

The interaction diagram in Figure 10.1 shows the somewhat simplified code path that is executed for one HTTP request.

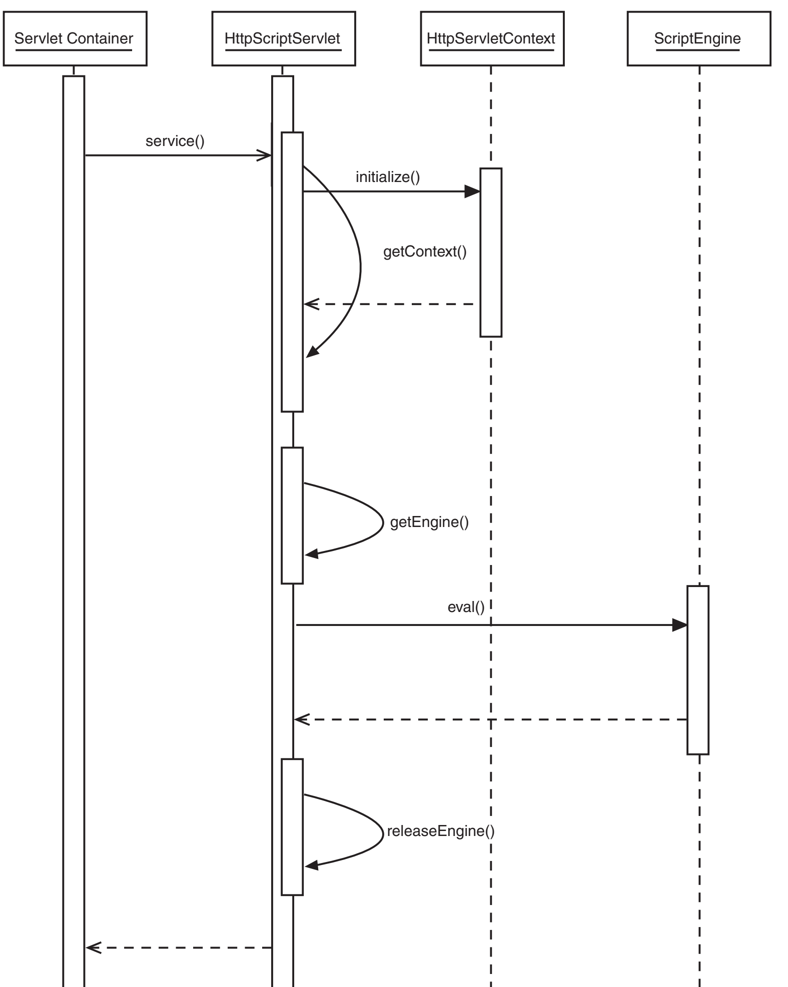


FIGURE 10.1 Web Scripting Framework request handling

As you can see from the interaction diagram in Figure 10.1, the client's requests are handled by the `service()` method (defined in the `javax.servlet.Servlet` interface). This method is implemented in the `HttpScriptServlet` abstract class, but it could be overloaded in its extensions. A default implementation calls the `getContext()` method that initializes the `HttpScriptContext` interface implementation.

After this step, an appropriate engine is pulled out according to the request (usually by the URL extension), and the appropriate script is evaluated in the previously initialized context. At the end, the engine is released, and the control is returned to the servlet container.

Getting Started

Now that we know the basic abstractions of this framework, and how they interact, I can go further and explain some configuration details and practical examples. To start, we have to enable support for the Web Scripting Framework in a servlet container or a particular Web application. This assumes that we have to register an instance of the `HttpScriptServlet` implementation inside the container and map it to handle desired URLs. There are two places where we can do this:

- We can define a servlet on the container level and make it available to all applications deployed in it. This configuration is server dependent, and you should consult your container's documentation for further details. For Tomcat, this is done in the `$TOMCAT_HOME/conf/web.xml` file.
- On the other hand, we can enable scripting on an application basis by registering an `HttpScriptServlet` implementation in the appropriate `WEB-INF/web.xml` file (the Web application deployment descriptor). In the rest of this chapter, we assume this approach and apply all further discussion of configuration details to the appropriate `WEB-INF/web.xml` descriptor.

In both cases, the appropriate packages must be available in the classpath. Generally, you'll need JSR 223 reference implementation (`script.jar`), JARs needed to properly run scripts of the desired scripting language that you want to use, and the appropriate script engine implementation. If you installed JSR 223 reference implementation by the Appendix C instructions, your Tomcat installation should be properly configured to run examples from this chapter.

There are two ways to achieve this. You can do it by placing the appropriate JARs in the `WEB-INF/lib` folder and making the folder available to a certain Web application, or by placing them in the container's classpath and making them available to all Web applications. Which approach you should take depends on whether you will make scripting generally available in the container.

Now look at the `web.xml` Web application descriptor shown in Listing 10.1, which is the entry to our sample Web application.

Listing 10.1 Sample Web Application Descriptor

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
 PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
 "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>

  <display-name>Scripting Web Application</display-name>
  <description>
    Demonstrates Web Scripting Framework (javax.script package)
  </description>

  <servlet>
    <servlet-name>ScriptServlet</servlet-name>
    <servlet-class>
      com.sun.script.http.ScriptServlet
    </servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>ScriptServlet</servlet-name>
    <url-pattern>*.php</url-pattern>
  </servlet-mapping>

  <welcome-file-list>
    <welcome-file>index.php</welcome-file>
  </welcome-file-list>

</web-app>
```

Here, we defined a servlet named `ScriptServlet` and implemented it through a default `HttpScriptServlet` extension class. Next, we mapped all URLs with the `php` extension (PHP scripts) to be handled by this servlet. That is basically it; we have enabled PHP scripts (or generally speaking, scripts written in any supported scripting language) to be executed inside our Java servlet container.

Now let's create a PHP script to test this configuration:

```
<html>
<head>
    <title>
        Web Scripting Framework
    </title>
</head>
<body>
Hello world<br>
Current time:
<?
    echo date('Y-m-d H:i:s');
?>
</body>
</html>
```

This simple script contains HTML page formatting and simple embedded PHP code that just prints the current date and time.

Suppose, for example, that our Web application is deployed on the `/script` path and that the script is named `index.php` and is located in the application's context root. If this were the case, it would be reachable via the following URL:

```
http://localhost:8080/script/index.php
```

The resulting output would be similar to that shown in Figure 10.2.

FIGURE 10.2 The result of execution of a PHP file in a servlet container

As you can see, the PHP script is executed in the same way as JSP pages or another application's servlets would be. With this configuration, we have extended the basic functionality of our servlet container.

Configuration

As we have seen thus far, there is no need for any additional configuration besides the standard servlet definition and mapping to enable Web Scripting Framework support in a servlet container or Web application. However, you can adapt it further through additional configuration parameters. Those parameters are related to the `HttpScriptContext` interface implementation and are defined as application context parameters using the `<context-param>` tags in the appropriate descriptor. In the following sections, I describe some parameters defined by the specification.

Disable Scripting

Even if we have the `HttpScriptServlet` implementation defined and mapped to handle certain URLs in a servlet container, we can disable scripting in some Web applications. We can do this through the `script-disable` Web application context parameter.

If you add the following snippet to the `WEB-INF/web.xml` descriptor from the previous example:

```
<context-param>
    <param-name>script-disable</param-name>
    <param-value>true</param-value>
</context-param>
```

and you try to execute the same URL:

```
http://localhost:8080/script/index.php
```

the server will return a response with the status code `403` (defined by the `HttpServletResponse.SC_FORBIDDEN` field), indicating that such an action is not permitted on the server.

This parameter is not valuable in situations where scripting is enabled on an application basis because in that case you control whether you will set that support in your application. But in situations where scripting is enabled for the whole container (as in an ISP hosted environment), you might need this parameter to disable scripting and thus prevent any possible malicious actions from occurring.

All the parameters that we cover in this section represent a configuration of the `HttpScriptContext` interface implementation. Initialization of this abstraction is done using the `initialize()` method, and values can be obtained by the appropriate methods. For example, the `service()` method of the `HttpScriptServlet` class uses the `disableScript()` method of the `HttpScriptContext` object to check the value of this configuration parameter and take the appropriate action.

Script Directory

By default, script resources are located, as are any other Web resources, in a Web application. That means paths to script files will be resolved starting from the root context of the Web application.

For example, suppose that our Web application is deployed to the Tomcat servlet container that is installed in the `/opt/tomcat` directory. Usually, Web applications are installed in the `/opt/tomcat/webapps` folder. Our application, as I already said, is installed in the `script/` context, so its context root will be the `/opt/tomcat/webapps/script/` directory. Thus, in our example, the following URL executes a script located in the `/opt/tomcat/webapps/script/index.php` file:

```
http://localhost:8080/script/index.php
```

The same mapping rules apply for all resources, including JSP and static HTML pages.

You might want to change these mapping rules, for whatever reason. For example, maybe you already have installed a PHP Web application in another folder, or you want to use security permissions on Groovy scripts.

If you want to change the directory location that contains all the scripts that can be evaluated in the Web application, you should use the `script-directory` context configuration parameter:

```
<context-param>
    <param-name>script-directory</param-name>
    <param-value>/opt/script/</param-value>
</context-param>
```

The previous example sets the `script-directory` parameter to the `/opt/script` directory, so now our example URL evaluates the `/opt/script/index.php` file.

If the specified directory does not exist or is inaccessible for any reason, the server will return a response with the status code 404 defined with the `HttpServletResponse.SC_NOT_FOUND` field.

Script Methods

Another parameter that we can control through these context configuration parameters is a list of HTTP methods that can be handled by scripts in a Web application. For example, you might want to allow scripts to handle only requests submitted using the `POST` method. For that purpose, you could use the configuration detail shown in the following code snippet:

```
<context-param>
    <param-name>script-methods</param-name>
    <param-value>POST</param-value>
</context-param>
```

As you can see, the `script-methods` parameter is used to control this behavior. Allowed methods are submitted as a comma-delimited list of HTTP request methods.

If a URL that is mapped to the `HttpScriptServlet` is requested using a method that is not supported, the server will return a response with the status code 405 (`HttpServletResponse.SC_METHOD_NOT_ALLOWED`). If this parameter is not defined, scripts can handle `GET` and `POST` methods. This is equivalent to the following configuration:

```
<context-param>
    <param-name>script-methods</param-name>
    <param-value>GET,POST</param-value>
</context-param>
```

Allow Languages

If there is no specific configuration, HttpScriptServlet could evaluate scripts in any language registered with its manager. To restrict the framework to work only with a subset of those languages, you should pass their names as the comma-separated list to the allow-languages configuration parameter. Names must match one of the values returned by the getNames() method of the appropriate ScriptEngineFactory class. An attempt to evaluate a script mapped to a language that is not allowed results in a response with the status code 403 (HttpServletResponse.SC_FORBIDDEN) returned to the client.

If we add the following configuration snippet to our example web.xml file and try to execute the example PHP script, the browser will indicate that the specified resource is forbidden:

```
<context-param>
    <param-name>allow-languages</param-name>
    <param-value>groovy</param-value>
</context-param>
```

The following configuration allows both Groovy and PHP (and only these two) language scripts to be executed:

```
<context-param>
    <param-name>allow-languages</param-name>
    <param-value>groovy,php</param-value>
</context-param>
```

Of course, to be able to run Groovy scripts, you have to make an appropriate mapping to the ScriptServlet.

```
<servlet-mapping>
    <servlet-name>ScriptServlet</servlet-name>
    <url-pattern>*.groovy</url-pattern>
</servlet-mapping>
```

Display Result

Some scripting languages are naturally embeddable into HTML pages (such as PHP, for example); others are not designed in that fashion. Later in this chapter, I address this issue in more detail, and we see how we can adapt general scripting languages to the Web environment. For now, let's focus on one more mechanism that could be helpful when using general-purpose scripting languages with the Web Scripting Framework.

You can generate Web content in languages that are not HTML embeddable in two ways. The first way is to use the language's `print` statement because the standard output will be flushed back to the client's browser. The second approach, which we discuss here, is to use a result of the script's evaluation as the content that should be returned to the client.

Consider the following Groovy script:

```
return """
<html>
<head>
    <title>Display result example</title>
</head>
<body>
This is a Groovy page that demonstrates the
<i>script-display-results</i> configuration property
</body>
</html>
"""
```

Now if we visit the following URL, a page like the one in Figure 10.3 will show up.

```
http://localhost:8080/script/groovy/display.groovy
```

As you have probably noticed, our script does not print any output. It simply returns a string value defined using the triple-quote syntax explained in Chapter 4, "Groovy." This is default behavior for the Web Scripting Framework; a result returned from the script evaluation (if any) will be embedded into the response (see Figure 10.3).

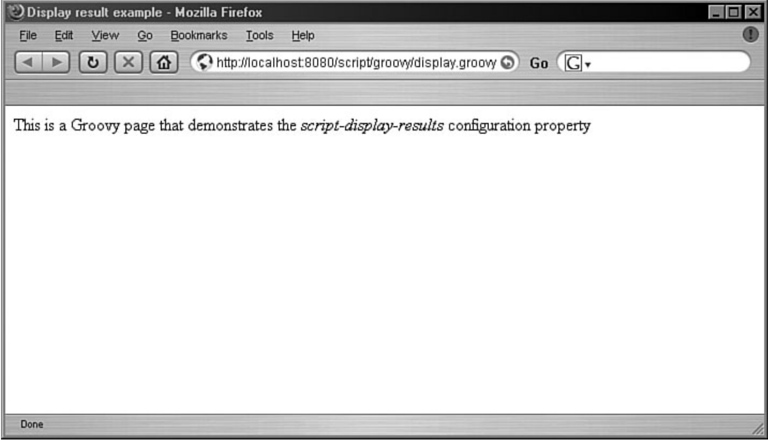


FIGURE 10.3 Display result enabled

In certain situations, however, we might want to prevent such behavior from occurring. For example, we might be using scripts developed for other environments and purposes. In those situations, there is probably no need to embed return values into the content.

To control this behavior, you can use the `script-display-results` configuration parameter:

```
<context-param>
    <param-name>script-display-results</param-name>
    <param-value>false</param-value>
</context-param>
```

The default value of this configuration parameter is `true`, which means return values will be embedded into responses. If you put the preceding configuration snippet into your `web.xml` Web application deployment descriptor, you will explicitly forbid this behavior from occurring.

The resulting page for this configuration looks like the one shown in Figure 10.4.

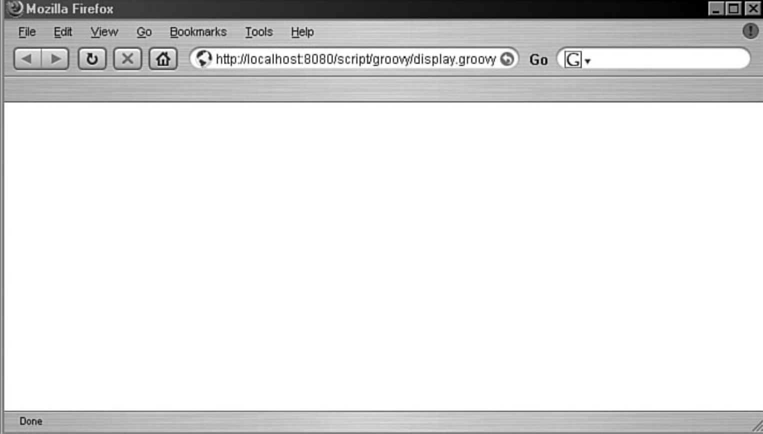


FIGURE 10.4 Display result disabled

Bindings

In the Scripting API discussion in Chapter 9, I explained how a context (an implementation of the `ScriptContext` interface) is mapped to the `context` script variable. I also said that in a servlet environment, scripting engines use the `HttpScriptContext` interface as a key abstraction to represent the script execution context. So it is natural to expect that script coders have the `context` variable available in their scripts. This variable is used to access attributes in the global and engine scopes, just as was the case with standalone script engines. This context also can be used to access attributes in the request, session, and application scopes. In other words, we can use these attributes to access states bound to a particular HTTP request, HTTP session, and servlet container in which the script is executed.

Let's walk through some examples and see how we can use these attributes.

Application

As I said, the application context is used to enable script engines to use attributes set by the servlet container in which

it executes scripts. What attributes will be set depends on the particular servlet container we use. For example, the Apache Tomcat container enables Web application developers to get a list of welcome files by using the `org.apache.catalina.WELCOME_FILES` attribute.

Suppose that in our `web.xml` configuration file, we have the following configuration snippet:

```
<welcome-file-list>
    <welcome-file>index.php</welcome-file>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

This means when the user hits a URL that refers to a directory, the container looks for a file defined in this configuration tag. If the file is found in the directory, it will be processed, and the result will be returned to the user. If no welcome files are found, the container will display a directory listing or return the 404 status code, depending on how it is configured.

Now look at the PHP script in Listing 10.2, which can access this configuration parameter if, of course, it is executed in the Tomcat servlet container.

Listing 10.2 Accessing Application Context

```
<html>
<body>
Welcome files:
<hr>
<?
    $welcome = $context->getAttribute(
        "org.apache.catalina.WELCOME_FILES"
      , $context->APPLICATION_SCOPE
    );
    foreach ($welcome as $file) {
        echo $file . "<br>";
    }
?>
</body>
</html>
```

This example is simple and straightforward. We used the `getAttribute()` method of the `HttpScriptContext` implementation and fetched a desired attribute. Because this attribute is a list, it was converted to the PHP array. Then we traversed it and printed out all its members.

If we now run this script by visiting the following URL, we will get a page like the one shown in Figure 10.5.

```
http://localhost:8080/script/application/application.php
```

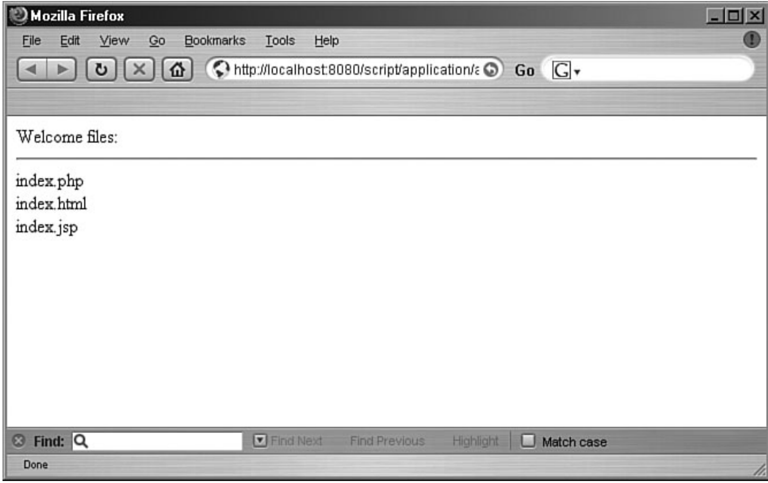


FIGURE 10.5 Welcome file list

Use of these servlet container attributes is not common in standard Web application development because these attributes are highly container dependent. But it can be useful in situations when your project is tied to a specific container and you need to act differently according to some of the configuration parameters. You also could use them for storing applicationwide attributes from scripts (however, an alternative storage approach is the use of servlet context initialization parameters, as we see in a moment).

Request

In the same manner as I just explained, we can access request attributes from scripts executed in the Web Scripting Framework. However, this approach is not flexible enough for Web application developers because, besides the attributes, they usually need access to other request information, such as the request URI, for example.

For that purpose, the `javax.servlet.http.HttpServletRequest` Java object, which stores all the necessary information on the particular request, is mapped to the `request` variable of the engine scope. In this way, all request details are available to script developers.

Let's go through a simple example and see how we can use this functionality to process HTML forms. First, we create a simple form (see Listing 10.3).

Listing 10.3 HTTP Request Handling—HTML Form

```
<html>
<body>
<form name="request" action="action.php">
    <table>
    <tr>
            <td colspan="2">Form</td>
    </tr>
    <tr>
        <td>Username</td>
        <td><input type="text" name="username"> </td>
    </tr>
    <tr>
        <td>Password</td>
        <td><input type="password" name="password"></td>
    </tr>
    <tr>
        <td colspan="2"><input type="submit" value="Login"></td>
    </tr>
    </table>
</form>
</body>
</html>
```

You could use a form such as this to log users onto your Web site. It consists of two fields, `username` and `password`, and a `submit` button. If you go to the following URL, a form similar to the one in Figure 10.6 will show up.

```
http://localhost:8080/script/request/login.jsp
```

As you can see from the script's source, the action URL for this form is the `action.php` script. Our example script that processes a request and parameters submitted through the form is shown in Listing 10.4.

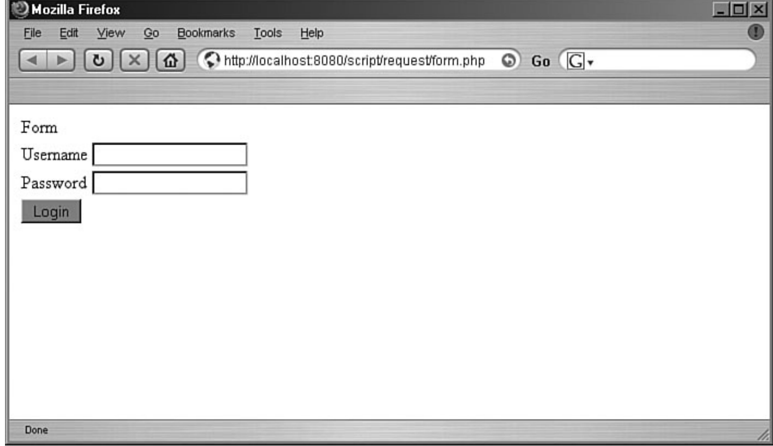


FIGURE 10.6 Form

Listing 10.4 HTTP Request Handling—PHP Script

```
<table border ="1">
<tr>
    <td colspan="2">Request parameters</td>
</tr>
<tr>
    <td>URI</td>
    <td><? echo $request->getRequestURI(); ?></td>
</tr>
</table>
<p>
<table border ="1">
<tr>
    <td colspan="2">Submitted attributes</td>
</tr>
<tr>
    <td>Username</td>
    <td><b><? echo $_GET["username"] ?></b> </td>
</tr>
<tr>
    <td>Password</td>
    <td><b>
        <?
            import_request_variables("gp");
            echo $password;
        ?>
        </b>
         
    </td>
</tr>
</table>
```

We can learn a few interesting things from this example. First, we used the `request` variable to access the request servlet object in the way that I explained at the beginning of this section. We called the `getRequestURI()` method to obtain the URI of this request (the part from the protocol name, up to the query string).

Next, we see that the reference implementation maps request variables to the standard PHP `$_GET` and `$_POST` variables. These variables are commonly used in PHP for these purposes, so this mapping makes servlet containers an even more natural environment for PHP developers.

Finally, after the `import_request_variables()` PHP function is called, we can use request attributes as PHP variables directly.

Now, if we fill in the form and click the Submit button, this PHP script will be executed and a page similar to the one shown in Figure 10.7 will be displayed.

FIGURE 10.7 Action PHP script

As we saw in this example, the Web Scripting Framework created an environment where scripts written in scripting languages could be used similarly to JSP pages. Also, the framework could be further expanded to support specific language features and thus provide an easy transition for developers coming from those backgrounds.

Response

In a similar manner to the one I described in the "Request" section earlier in the chapter, we can use the response servlet object as a script variable. For example, we can use it to set a custom status code for our response.

For instance, let's extend the example shown in Listing 10.4 and return the status code 403 (forbidden access) if the user doesn't provide his username or password. For that purpose, we add the following code snippet at the beginning of our preceding example:

```
<?
    import_request_variables("gp");
    if ($username == '' || $password == '') {
        $response->sendError($response->SC_FORBIDDEN);
    }
?>
```

Here, we tested the `username` and `password` request attributes, and if any of them is empty, we used the `response` servlet object to send a response with the appropriate status. For that purpose, we used the `sendError()` method and the `SC_FORBIDDEN` field of the `javax.servlet.http.HttpServletResponse` interface implementation in the appropriate servlet container.

Servlet

For even more script programming convenience, the servlet object itself is bound to the engine scope. This means we can freely use its methods for various tasks. We can, for example, use the `log()` method and log a message directly in the servlet's log file. Let's demonstrate this functionality by extending our previous response example (we add the line marked as bold):

```
<?
    import_request_variables("gP");
    if ($username == '' || $password == '') {
        $servlet->log("credentials are not provided");
        $response->sendError($response->SC_FORBIDDEN);
    }
?>
```

Now, if the user does not provide his username or password, the message will be logged to the servlet log, and the status code 403 (forbidden) will be returned. The location of the log file depends on the container and Web application configuration. For the Tomcat server, it is usually located in the $TOMCAT_HOME/logs directory. If you now submit the form with either the username or the password field left blank and you look at the appropriate log file, you should see a message similar to the following:

```
2005-05-07 02:15:25 StandardContext[/script]ScriptServlet:
  credentials are not provided
```

In a similar manner, we can access other public methods of the javax.servlet.http.HttpServlet class and get initialization parameters, information, and so on.

Include Method

To enable a fully operable Web development environment, we have to enable script developers to include other resources contained in the Web server. For this task, the javax.script.HttpScriptContext interface defines the include() method.

This method accepts a string parameter that represents a relative URL path to the resource that has to be included. If the path starts with the / character, the absolute URL is calculated from the context root of that application. In any other case, the absolute URL will be calculated (relative to the URL of the script). This is the same rule as the one used in the include() method of the javax.servlet.jsp.PageContext class. This class plays a similar role in the JSP environment as the HttpScriptContext interface in the Web Scripting Framework.

Now, we can use this knowledge to create the following example. Imagine that you have a Web application created with JSP technology. This application has a unified header and footer for all pages. The header is defined in the _header.jsp file and is included at the beginning of every page.

```
<html>
<head>
    <title>Include demo</title>
</head>
<body>
<table width="80%">
    <tr>
        <td align="left">
            Scripting in Java
        </td>
        <td align="right">
            <%= new java.util.Date() %>
        </td>
    </tr>
</table>
<hr>
<table width="80%">
```

In the same manner, the footer is defined in the _footer.php file and is included at the end of each page.

```
</table>
<hr>
<table width="80%">
&copy; Dejan Bosanac
</table>
```

Now, we want to add a PHP script and make it an integral part of this application. We also want to preserve the original header and footer. To achieve all this, we can use the previously described include() method to include the _header.jsp and _footer.jsp files in this script, as shown in Listing 10.5.

Listing 10.5 Including Resources

```
<?
    $context->include('./_header.jsp');
?>
Page content
<?
    $context->include('./_footer.jsp');
?>
```

As I already said, the HttpScriptContext object is bound to the engine scope's context variable. So, in this example, we included the header and footer files that are located in the same directory as our script.

If you run this example by visiting the following URL, you can expect to see the page shown in Figure 10.8.

```
http://localhost:8080/script/forward/include.php
```

FIGURE 10.8 Include demo

Forward Method

Besides including other resources, Web developers often need to redirect, or forward, a request to another resource. For that purpose, the `HttpScriptContext` interface defines the `forward()` method.

As with the `include()` method, the `forward()` method takes a relative URL path as an argument. The rules used for calculating an absolute URL of the target resource are also the same as those for the `include()` method.

As an example of the `forward()` method, we can use our login form again, as shown in Listing 10.6.

Listing 10.6 Request Forwarding

```
<?
    import_request_variables("gP");
    if ($username == '' & $password == '') {
        $servlet->log("credentials are not provided");
        $context->forward("login.jsp");
    }
?>
```

Now, instead of sending a response with the status code 403 to the user, we redirected the user to the `login.jsp` page and showed him the login form again.

Now, let's take a minute to understand the differences between the `include()` and `forward()` methods because they can be confusing for developers new to Web applications.

When the `include()` method is used, the output generated before the method call is flushed to the browser. Also, the control is returned to the script after the external resource is processed. So if we define a simple `index.jsp` resource like this:

```
Included text
```

and then create the following script:

```
Text before
<br>
<?
    $context->include('index.jsp');
?>
<br>
Text after
```

we will have output similar to that shown in Figure 10.9 as a result of the script evaluation.

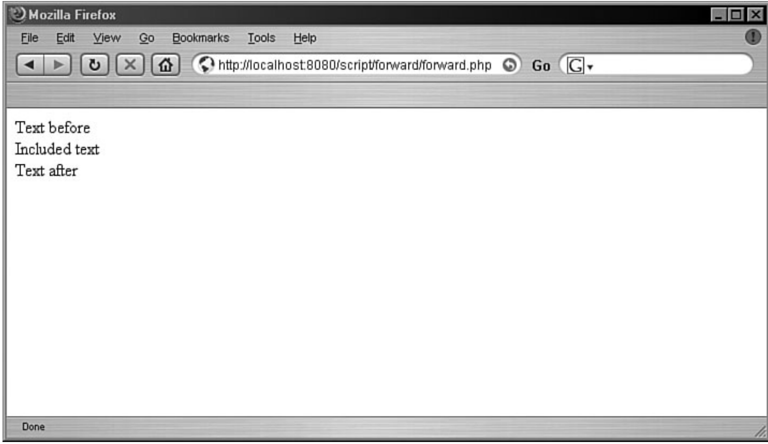


FIGURE 10.9 Include

However, the content generated before the `forward()` method call is discarded, and the control is not given back to the original script. So if we change the preceding example to use the `forward()` method instead:

```
Text before
<br>
<?
    $context->forward('index.jsp');
?>
<br>
Text after
```

we will get the result shown in Figure 10.10.

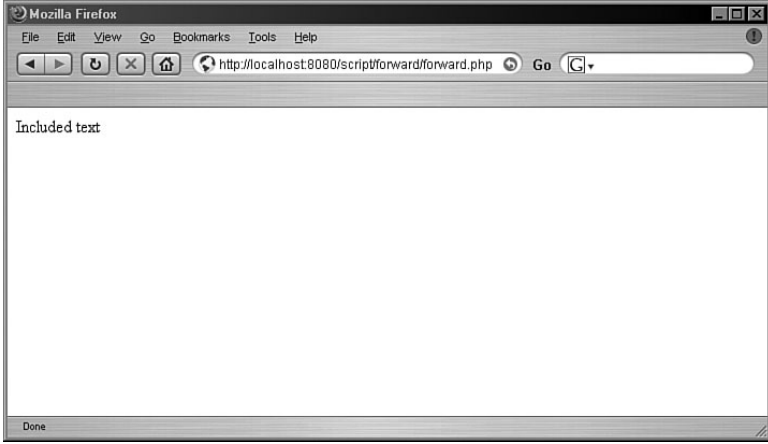


FIGURE 10.10 Forward

Developers need to know this difference to make the right choice for their solutions.

Session Sharing

User session handling is a crucial task in dynamic Web application development. Because the HTTP protocol is stateless, a Web server (or a servlet container in our case) cannot save user-related data between two requests. Every request is completely independent of the previous one, so the session mechanism was developed to fill this gap.

Before we discuss the session handling mechanism implemented in the Web Scripting Framework, let's talk about how sessions work. Session data is stored on the server side, inside the Web server. Two approaches are commonly used to map a session to a user.

The first approach is to send a cookie to the user. That cookie has a reserved name (that could be configured in the server) and contains a unique session identifier for that user. The second approach is to send this unique session ID as a request parameter. You would usually do this to support browsers that don't support cookies for whatever reason. Which of these two methods you use usually depends on how the server is configured.

When the user submits a request to the Web server, the Web server looks up the session ID, and if it finds it, the session data is mapped to that request. From that point on, the servlets and scripts in the Web application can use sessions to store data that has to be persistent among several requests.

The most common use of sessions is for user authentication purposes, so our example will handle that case. Imagine that you have a Java Web application and you want to integrate some PHP code in it. Suppose that user sessions will be initialized in the Java part of the application and that the data should be accessible in PHP scripts as well.

The first thing we have to do is create a simple login form, as shown in Listing 10.7.

Listing 10.7 Session Handling—Login Form (`form.jsp`)

```
<form name="request" action="login">
    <table border ="1">
    <tr>
            <td colspan="2">Form</td>
    </tr>
    <tr>
        <td>Username</td>
        <td><input type="text" name="username"> </td>
    </tr>
    <tr>
        <td>Password</td>
        <td><input type="password" name="password"></td>
    </tr>
    <tr>
        <td colspan="2"><input type="submit" value="Login"></td>
```

Listing 10.7 Continued

```
    </tr>
    </table>
</form>
```

This form is practically the same as the forms we used in our preceding examples. The only thing that is important to note is the action URL because we will map that URL to our login servlet.

Now let's create a servlet that handles login operations in the application (see Listing 10.8).

Listing 10.8 Session Handling—Login Servlet

```
package net.scriptinginjava.ch10.web;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class LoginServlet extends HttpServlet {


    protected void service(HttpServletRequest request,
        HttpServletResponse response)
            throws ServletException, IOException {

        String username = request.getParameter("username");
        String password = request.getParameter("password");

        if (username.equals("") || password.equals("")) {
           response.setStatus(HttpServletResponse.SC_FORBIDDEN);
           return;
        }

        // check user credentials in the database

        request.getSession().setAttribute("user", username);
        response.sendRedirect("index.php");
    }
}
```

This servlet is simple, and we use it here only for demonstration purposes. First, we obtained the `username` and `password` parameters from the request (submitted from the form). If these parameters are not found, we will return the

status code that indicates that access to the resource is forbidden. We used this approach in all our previous examples, but in a real application, you more likely will redirect your client to the page with an appropriate message.

After parameter checking is done, we have to check whether the user has submitted valid credentials. This code is purposely omitted because it is not important for our example, and the specific implementation depends highly on the technology you are going to use in your project.

After the validation is finished, we start the user session and set the username as a `user` parameter. Finally, we redirect the user to the `index.php` page, and because it has the `.php` extension, it is processed with our `ScriptServlet` defined in our previous examples.

To run this example, we have to compile this servlet first. For that we need to have Servlet API in the classpath. You can usually find it in the servlet container you are using. For example, in Tomcat you can find an appropriate JAR in the `common/lib/` folder. When compiled, this class should be placed in the `WEB-INF/classes/net/nighttale/ch10/web` folder of our application, so that server could use it.

> **Note**
> The session examples are placed in the `session` subfolder, so this path info must be present in the `url-pattern` configuration parameter.

Now we have to map this servlet to process the URL defined in the form's `action` parameter. We do this by putting the following code snippet into the Web application deployment descriptor (`web.xml`):

```
  <servlet>
      <servlet-name>LoginServlet</servlet-name>
      <servlet-class>
          net.scriptinginjava.ch10.web.LoginServlet
      </servlet-class>
  </servlet>

<servlet-mapping>
      <servlet-name>LoginServlet</servlet-name>
      <url-pattern>/session/login</url-pattern>
</servlet-mapping>
```

Now we can write the `index.php` script and see how we can use session data in scripts (see Listing 10.9).

Listing 10.9 Session Handling—PHP Script

```
<?
  $username = $request->getSession()->getAttribute("user");

  // $username = $_SESSION["user"];
  // not currently implemented

  if ($username == "") {
      $context->forward("form.jsp");
  }
?>

Hello <?= $username ?>
```

Because the `javax.servlet.http.HttpServletRequest` object is mapped to the `request` script variable, we can use it in a standard Java fashion. To get an instance of the `javax.servlet.http.HttpSession` object, we have to call the `getSession()` request method. The `getAttribute()` method of the session object returns the appropriate attribute.

After we fetch the `user` attribute, we should test whether it has been set. If it hasn't, it means the user is not logged in, and we have to redirect him back to the login form. We show him the welcome message, otherwise.

> **NOTE**
> In this simple example, we used a simple string value, but as we know, we could share any Java object in this way between the servlet container and scripts.

Unfortunately, the session object is currently not mapped to the `$_SESSION` PHP variable, which is the place where PHP developers usually look for session attributes. This would increase portability a great deal, and it will probably be the subject of future improvements.

In some situations, you may want to forbid scripts from accessing session data. You would typically do this because of some security concerns that you might want to apply in your system. You should use the `script-use-session` context configuration parameter to control this property. Here is an example configuration snippet:

```
<context-param>
    <param-name>script-use-session</param-name>
    <param-value>false</param-value>
</context-param>
```

The default value of this property is `true`, which means the scripts can use session objects freely. If the value is set to

`false`, all methods that work with the `HttpSession` objects will be forbidden. This means

- The `getSession()` method will return `null` for every request.
- The `getAttribute()` method of the `HttpScriptContext` objects will always return `null` for `SESSION_SCOPE` attributes.
- The `setAttribute()` method of the `HttpScriptContext` objects will throw an `IllegalArgumentException` for `SESSION_SCOPE` attributes.

Language Tags

We wrote most of our examples thus far in the PHP programming language. As we know, this language is naturally embedded into HTML pages. This means scripts are embedded into HTML pages within `<? . . . ?>` blocks, so when the server handles a resource, it extracts those blocks and evaluates them using the appropriate scripting engine. The content generated by the script is then put back into the HTML page, and the page is returned to the client.

This is important for Web developers because it eases content generation a great deal. Prior to this approach, development of dynamic Web applications was related to execution of programs (or scripts) that used their print statements to generate all the content shown on a page. This can be a cumbersome thing to do because pages are mostly static, with only some dynamic elements. Consider, for example, a page's header and footer. Logo information and copyright notices are not likely to be dynamic, so making them a part of the script that will be evaluated makes the solution harder to create, read, and maintain.

Also, usually different parties are involved in the application's development. The two most common roles are those of Web designers, who are responsible for the page design and the overall look of the application, and Web programmers, who

create application logic and work on pages in other ways. The existence of tags allows a good separation of concerns so that both designers and developers can work on the same resources without interfering with each other's work.

JSP is one of the technologies that solve the same problem for Java Web application developers. It allows Java code to be embedded into HTML pages and executed the moment that page is requested.

This works fine for PHP and other technologies created for Web development, but not all scripting languages were originally designed for this environment. So, to use them efficiently for generating Web content, we need a mechanism that makes them embeddable into HTML pages.

The mechanism shown here is not part of the official Web Scripting Framework specification. It is implemented in its reference implementation and could be useful for full Web support of a wide range of scripting languages.

Imagine that you want to use Groovy along with the Web Scripting Framework to create a simple page for your Web project. With all we have learned thus far, we would probably end up with a script that looks like the one in Listing 10.10.

Listing 10.10 Language Tags—Original Groovy Script

```
println "Hello world<br>"
println "Current time: "

date = new java.util.Date()
print date
```

This simple script prints two static text lines (marked in bold) and then dynamically inserts the current date (Java object). If you visit the following URL, you should get a page similar to the one shown in Figure 10.11.

```
http://localhost:8080/script/groovy/index.groovy
```

The problem with this approach is that this script is not very readable for a Web designer. Imagine that a Web designer needs to put a header and formatting details into this page.

Making him write `println` statements, worry about proper escaping, and so on, makes this process longer and much more prone to errors.

FIGURE 10.11 Untagged Groovy script

We need a way to define Groovy scriptlets within tags and leave all other content static. For that purpose, the `com.sun.script.http.ScriptServlet` implementation of the `HttpScriptServlet` interface (a reference implementation) defines another configuration parameter that we can use.

The `script-blocks` servlet initialization parameter takes a comma-separated list of languages, whose resources will be treated as HTML pages, with scriptlets defined inside the `<% . . . %>` tag blocks.

Listing 10.11 shows an example configuration of `ScriptServlet`.

Listing 10.11 Language Tags—Configuration

```
<servlet>
    <servlet-name>ScriptServlet</servlet-name>
    <servlet-class>
        com.sun.script.http.ScriptServlet
    </servlet-class>
    <init-param>
        <param-name>script-blocks</param-name>
        <param-value>js,groovy</param-value>
    </init-param>
</servlet>
```

As you can see, we instructed a servlet to treat JavaScript and Groovy resources as regular Web pages. The same rules as those for the context initialization parameter named `allow-languages` apply here, concerning names of languages that we will use.

Now, we can modify the script shown in Listing 10.10 to make it a more natural fit for the Web Scripting Framework (see Listing 10.12).

Listing 10.12 Language Tags—Groovy Script

```
Hello world<br>
Current time:
<%
    date = new java.util.Date()
    print date
%>
```

The behavior of the script remains the same. The only difference is that static content is now separated from scriptlets, which makes development and maintenance much easier to organize.

Threading Issues

In the servlet container, script engines used to handle script resources work in a multithreaded environment. This means scripts are executed concurrently on different threads.

In Chapter 9, we discussed various approaches to threading that scripting engine implementations could take. We also saw how we can find out what threading capabilities a certain scripting engine implements, by making the following call on the engine's factory objects:

```
ScriptEngineInfo.getParameter("THREADING")
```

We have seen that in the Web Scripting Framework a lot of important data, such as request and session parameters, is bound to the engine scope. With this in mind, we know that only script engines that satisfy the `THREAD-ISOLATED` concurrency requirement can be used freely in this environment. This

is simply because this requirement guarantees that engine scope will stay unchanged between two evaluations of the script, which is of course necessary to preserve consistency of a certain request state.

The script engine that will be used to handle a URL mapped to the script resource is obtained by the `ScriptEngine getEngine(HttpServletRequest request)` method defined in the `javax.script.http.HttpScriptServlet` interface.

Implementations of this interface must ensure that a scripting engine returned by this method is either `THREAD-ISOLATED` or that it currently does not execute any requests to preserve consistency in the Web application.

Architectural Challenges

Thus far, we learned the basic concepts built into the Web Scripting Framework. Now let's discuss where this technology would be most useful in practice. You can use the Web Scripting Framework in three areas of Web development:

- Integration of Java and PHP applications
- Migration of PHP application business logic to Java
- Use of PHP (or some other scripting language) as a view technology in your Java Web application

In this section, we examine these three use cases and discuss the implications they introduce.

Integration of Java and PHP Applications

First, many of you might ask why we would ever need to integrate Web applications created using different technologies. Although doing so is not a necessity, it could be convenient in many cases. For example, imagine that you are building a Web site portal using a content management system (CMS) product. Also, your customer wants to have a forum, blog, and who knows what else on that site.

One path that you can take is to develop all these components by yourself using the same technology. In this situation,

there are no integration issues, but the development costs (and time) would be much greater than they would if you used a variety of different technologies, each one developed to handle a specific task.

The second approach is to look for products (Web applications) that implement the components that you need. They could be either commercial products or open source projects. It depends completely on your requirements and on whether a certain product fulfills those requirements. Of course, if you want to create a pure Java (or pure PHP) solution, you have to consider the technology that lies behind the project that you want to incorporate.

Now that you have this standardized framework for integration of what today are the two most popular technologies for Web development, you have a much wider choice of projects that you can use. For example, you may find that CMS written in PHP suits your needs best, while you want to pick a Java project for a forum software. This broad array of choices can affect the overall quality of your final solution.

For successful integration of Web applications written for different environments, the crucial issue is sharing of session data. Prior to the Web Scripting Framework, sharing data between Java and PHP was possible, but it was much more complicated than it is today. One of the solutions was to implement your own session mechanisms for both Java and PHP Web servers and make them use the same data structures (such as files or databases, for example). Then you would be able to install some parts of your application on the servlet container and other parts on the Web server that would run a PHP interpreter (Apache Web server in most cases). Still, this solution is not as simple as it might look at first glance, and the maintenance of such a site would require administration of two Web servers rather than one.

The Web Scripting Framework has solved this problem in a much more natural way. Because now PHP scripts are evaluated in the servlet container environment and session data is bound to the engine, there is no need for solution developers to think about replacing a crucial part of their infrastructure software.

Even with all the concepts introduced by this framework, you still have to customize certain projects to make them integrate. The most common areas you would have to customize are the login and authentication code because these code parts are responsible for making a unique session across the applications.

Although these issues are beyond the scope of this book, we can freely say that the Web Scripting Framework is a great leap forward in terms of making these integrations possible, and that it will be the subject of future research.

Java Business Logic in PHP Web Applications

PHP and scripting concepts in general are excellent tools for rapid prototyping of Web applications. Thus, many developers use these tools to start a project and to implement initial requirements quickly.

As the project grows and becomes more heavily loaded, developers might find it appropriate to move some of the business logic code to the Java platform. This could benefit the project in general in many ways. Because Java is not tied to the Web server, architects and developers have a much wider range of solutions available to improve the scalability of the project.

Here, we discuss some of the issues related to moving part of a PHP project to Java. First, we assume that the PHP project was developed using the MVC (Model-View-Controller) design pattern. This means the business logic is clearly separated from the presentation, and no business code is present in presentation templates. (We will not discuss the MVC pattern in more detail here; you are advised to consult the appropriate literature for more information on that topic.) If the project does not apply the MVC pattern, our first step will be to refactor it toward this architecture.

Let's suppose we have the PHP class shown in Listing 10.13, and we want to replace it with its equivalent Java implementation.

Listing 10.13 PHP to Java—PHP Class

```
<?

class User {
```

Listing 10.13 Continued

```
    var $username;
    var $password;

    function verifyPassword($password) {
        if ($password == $this->password)
            return true;
        else
            return false;
    }
}

?>
```

The class is pretty simple. We have two properties and a `verifyPassword()` method. This method checks whether the submitted password is the same as the `password` field of the object.

Now let's suppose we have a factory class that is responsible for making instances of the `User` class. With this approach, the code for creating object instances is located in one place. As we see in a moment, a solution such the one shown in Listing 10.14 leads to much easier refactorings.

Listing 10.14 PHP to Java—Factory Class

```
<?

class UserService {

    function findUserByUsername($username) {
        $user =  new User();
        $user->username = $username;
        $user->password = $username;
        return $user;
    }
}

?>
```

The `UserService` class has a `findUserByUsername()` method, which in our example simply returns a `User` class instance. In a real application, this method would probably look for the user based on the given username in a database or some other repository (LDAP, for example). The username and password properties would also be populated with the appropriate fields of the database record. For the demonstration purposes of

this example, this simplified implementation is more than enough.

Finally, we need code that uses these “infrastructure” objects. Let’s say you are creating the login functionality for your Web site, as shown in Listing 10.15.

Listing 10.15 PHP to Java—Client Code

```
$user = UserService::findUserByUsername($username);

if ($user == null) {
    // reload the form and show the appropriate message
}

if (!$user->verifyPassword($password)) {
    // reload the form and show the appropriate message
}

// proceed with login action
```

The code in Listing 10.15 looks for the existence of the user with the provided username. If it does not find this user, the login form is reloaded, and the appropriate message is displayed. Next, we check whether the submitted password matches the user’s password. The form is reloaded if the password does not match. Otherwise, we proceed to other actions that are needed to finish the login.

Now let’s implement the Java equivalent of the `User` class (see Listing 10.16).

Listing 10.16 PHP to Java—Java Class

```
package net.scriptinginjava.ch10.domain;

public class User {

    public String username;
    public String password;

    public boolean verifyPassword(String password) {
        if (this.password == password)
            return true;
        else
            return false;
    }

}
```

The class implementation is straightforward and pretty self-explanatory, so we skip to the next step to finish the transition (see Listing 10.17).

Listing 10.17 PHP to Java—Modified Factory Class

```
<?
class UserService {

  function findUserByUsername($username) {
      $user =  new Java("net.scriptinginjava.ch10.domain.User");
      $user->username = $username;
      return $user;
  }
}
?>
```

As you can see, we changed only one line of the `UserService` class and substituted the PHP class with the Java implementation. From now on, the `User` class can use all the benefits of the Java platform and products made for it.

You can benefit from Java classes in your PHP Web applications in one more case. The life cycle of a PHP script is tightly coupled to handling a single HTTP request. In such an environment, it is hard to implement logic that needs some threading or scheduling capabilities. Now, you can delegate these tasks to Java threads and schedulers, and thus take more control over that process.

PHP Views in Java Web Applications

In the previous section, we saw how PHP Web applications can benefit from the Java platform. Now we discuss possible uses of PHP in Java Web applications.

Most Java Web applications are built using some of the available MVC frameworks, such as

- The Spring framework (www.springframework.org)
- Struts (http://struts.apache.org)
- WebWork (www.opensymphony.com/webwork/)

In all these frameworks, the view is usually created using some of the template engines available today. We discussed the

template mechanism back in Chapter 5, "Advanced Groovy Programming," where we saw some benefits of its introduction in the project. The Java community has a wide range of template languages available for developers to use. The most frequently used engines today are certainly the following:

- Velocity (http://jakarta.apache.org/velocity/)
- FreeMarker (http://freemarker.sourceforge.net/)

If you are working on a team that has a good PHP background and you are starting a Java Web project, you may consider using PHP as a template language for the project. Java Web applications introduce many technologies that a team has to cope with, such as the MVC framework, Object Relational mapping framework, and so on. By using PHP as a template language for the project, you can put off (or at least delay) having to learn one more technology at the start. The team's solid PHP background could speed up development and make them feel at home.

Particular solutions for creating PHP as a view technology in Java Web applications are currently beyond the scope of this book and will be the subject of future research and enhancements.

Conclusion

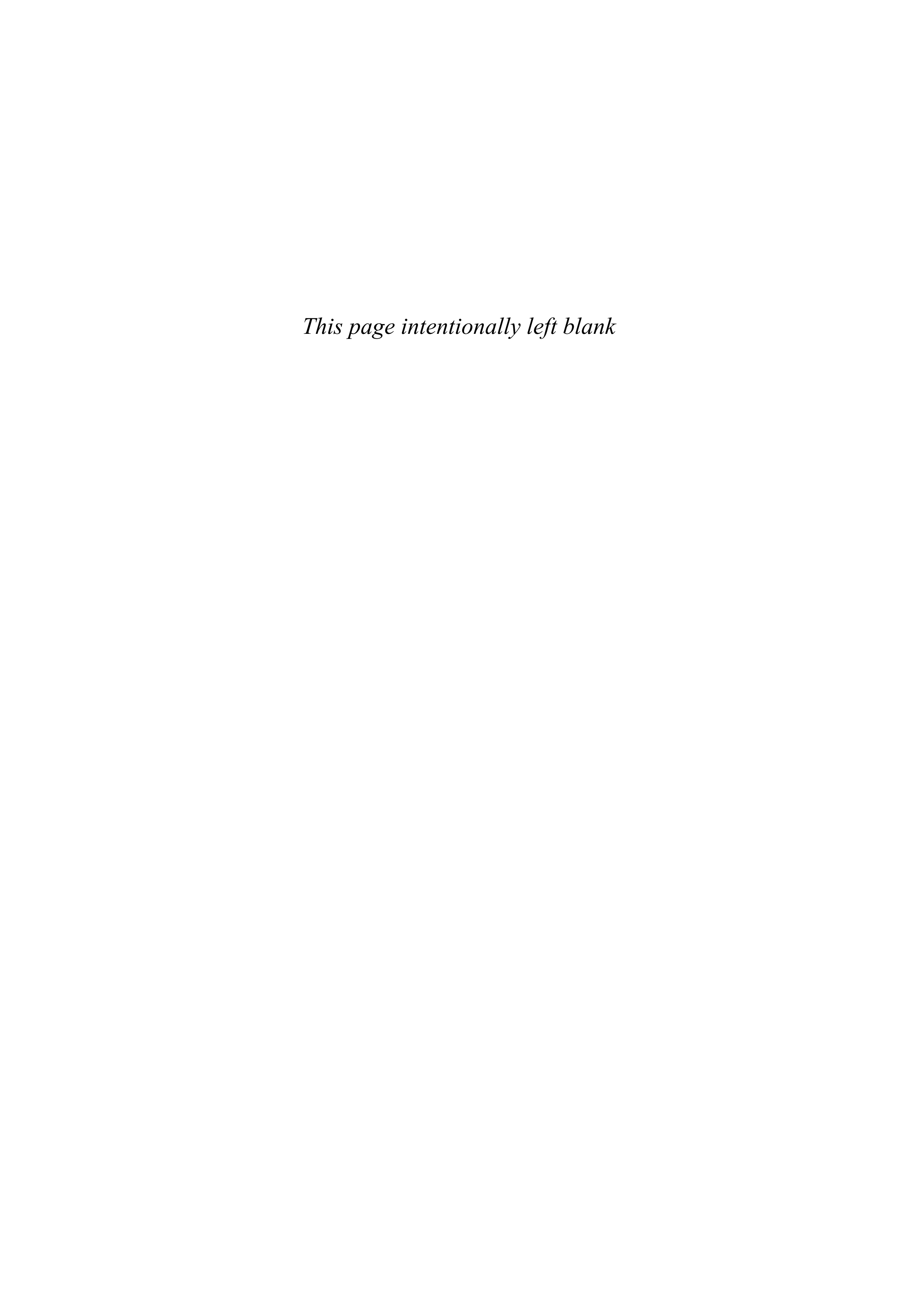
Throughout this chapter, we discussed the basic concepts of the Web Scripting Framework. We also learned about some basic ideas concerning the integration of two distinctive platforms, Java and PHP, by using the Scripting API and the Web Scripting Framework. We looked at it from both sides and emphasized the benefits of solutions in various environments.

Still, this is a relatively new topic, and I expect that many new projects and papers will target this field in the near future.

APPENDIX A Groovy Installation

APPENDIX B Groovy IDE Support

APPENDIX C Installing JSR 223



This page intentionally left blank

Appendix A

Groovy Installation

This appendix contains information necessary for successfully installing and configuring Groovy in your development environment. The first step is to download Groovy as a distribution, or to get the Groovy source code if you want to build it.

Download Instructions

The official Groovy site is located at http://groovy.codehaus.org. You can download the latest distribution by following the Download link on the home page, or by visiting http://dist.codehaus.org/groovy/distributions/. This URL contains both the source code and the binary distributions.

If you just want to embed Groovy in your application, you do not need the whole environment. The JAR (Java Archive) files with the Groovy implementation are located at http://dist.codehaus.org/groovy/jars/.

Installing Groovy

If you have downloaded the binary distribution, the installation procedure consists of unpacking the distribution archive.

Configuring Groovy

The steps needed for Groovy configuration are simple and straightforward:

1. Set the GROOVY_HOME environment variable to point to the Groovy installation—for example:

   ```
   Unix: export GROOVY_HOME=/opt/groovy
   Windows: set GROOVY_HOME=C:\groovy
   ```

2. Add the $GROOVY_HOME/bin directory to your PATH environment variable:

   ```
   Unix: export PATH=$PATH:$GROOVY_HOME/bin
   Windows: set PATH=$PATH;$GROOVY_HOME\bin
   ```

Testing Groovy

At this point, you should be ready to test the Groovy installation and write the “Hello world” script. Run the interactive Groovy shell by typing the following in the command line:

```
groovysh
```

You should get something similar to this on your display, depending on the versions of Groovy and Java that are installed on your system:

```
Let’s get Groovy!
================
Version: 1.0-JSR-06 JVM: 1.5.0_08-b03
Type ‘exit’ to terminate the shell
Type ‘help’ for command help
Type ‘go’ to execute the statements

groovy>
```

In this interpreter, you can write Groovy statements and execute them with `go` or `execute` commands. Now type:

```
groovy> println "Hello world!"
groovy> go
```

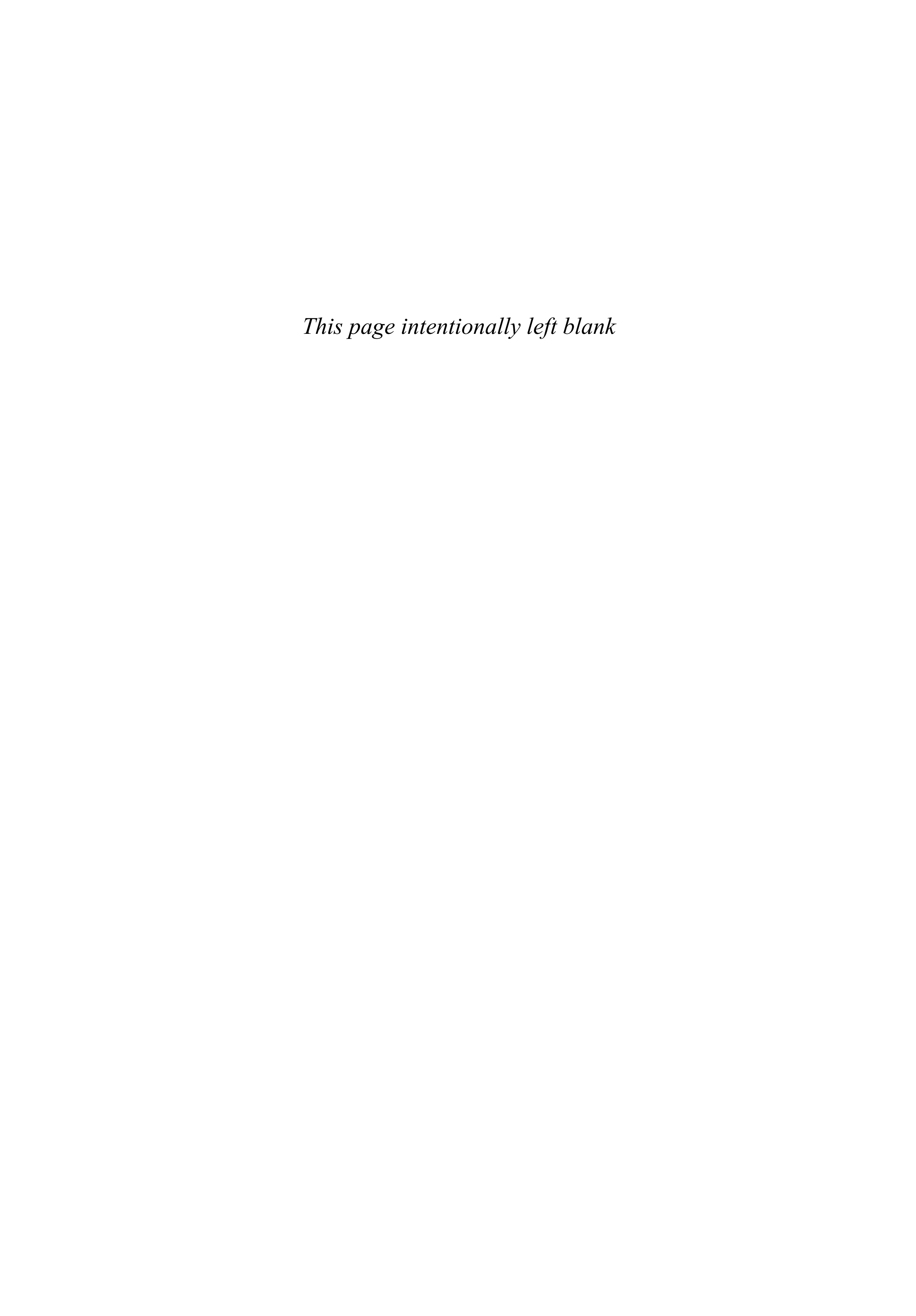
You should get the following text displayed on the screen as a result of statement's execution:

```
Hello world!
```

If you want to exit the interactive shell, just type:

```
exit
```

If you finished this little test without any errors, your Groovy environment is successfully installed.



This page intentionally left blank

APPENDIX B

GROOVY IDE SUPPORT

This appendix contains instructions on how to install general Groovy support for any Java editor or IDE. Most of the IDEs today either have built-in Groovy support or support it through the plug-in mechanism. For more information on this topic, consult the official Groovy Web site. The installation of general-purpose IDE support is demonstrated for the Eclipse platform.

Installation

As Groovy (and scripting in general) is becoming more popular among Java programmers, I am sure that all vendors and projects that build environments and editors for Java developers will support these technologies. But even if you cannot find the appropriate plug-in for your IDE (or editor), there is a way to evaluate Groovy scripts in any environment that can run Java classes. For that purpose, you should use the `groovy.lang.GroovyShell` class. For the `first.groovy` script example, the call would be

```
java groovy.lang.GroovyShell first.groovy test 123
```

> **NOTE**
> To execute this class, the appropriate `groovy` and `asm` JARs must be in the classpath. These JARs are located in the `lib/` directory of the Groovy distribution. You can find more information on this subject in the "Dependencies" section in Chapter 4, "Groovy."

I demonstrate this technique within the Eclipse platform. You should take similar steps in other environments to enable the `groovy.lang.GroovyShell` class to execute your Groovy scripts.

1. Select the Run option from Run menu, which starts a *Run configuration manager.*
2. Click the New button.
3. Enter `groovy.lang.GroovyShell` as the Main class (see Figure B.1). It is assumed that you have a Groovy JAR in the classpath of your project.

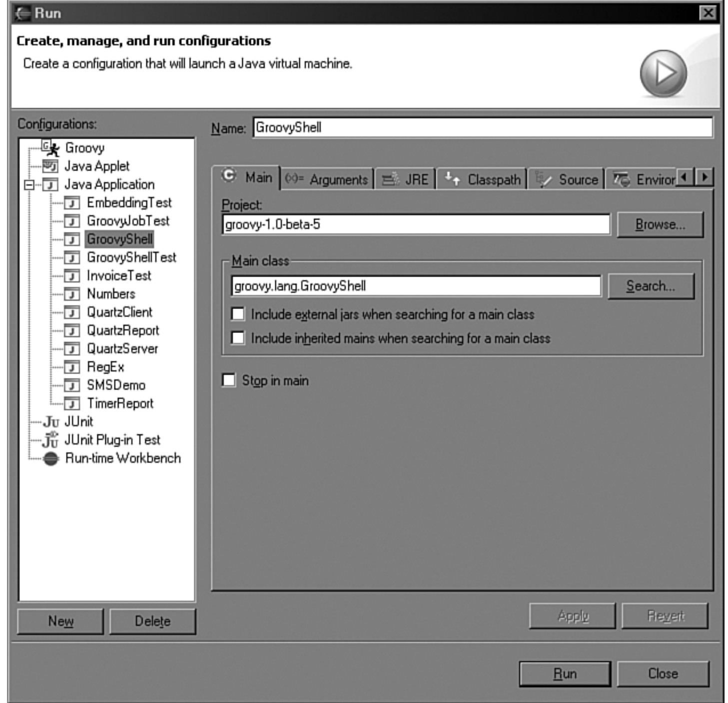


FIGURE B.1 Run configuration manager in Eclipse

4. In the Arguments tab, enter the `${resource_loc} ${string_prompt}` value in the Program Arguments section. It passes the absolute path of the selected script and prompts a dialog for entering arguments to be passed (see Figure B.2).
5. In the Classpath tab, add the appropriate `groovy`, `asm`, `antlr`, and `commons-cli` JARs to the classpath.

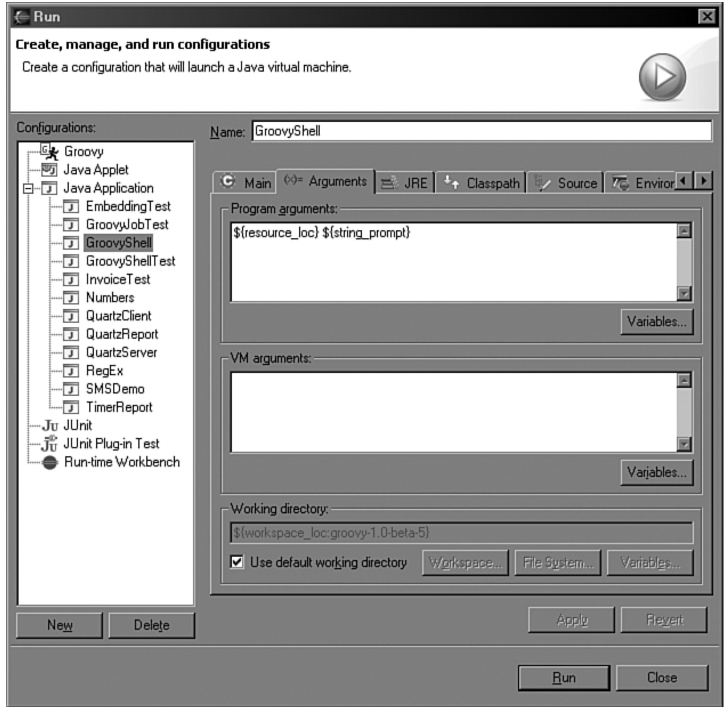


FIGURE B.2 Arguments tab

6. If you want to display this configuration in the Favorites menu, go to the Common tab and check the Run option. Save the changes that you made by clicking the Apply button and then close the configuration manager.

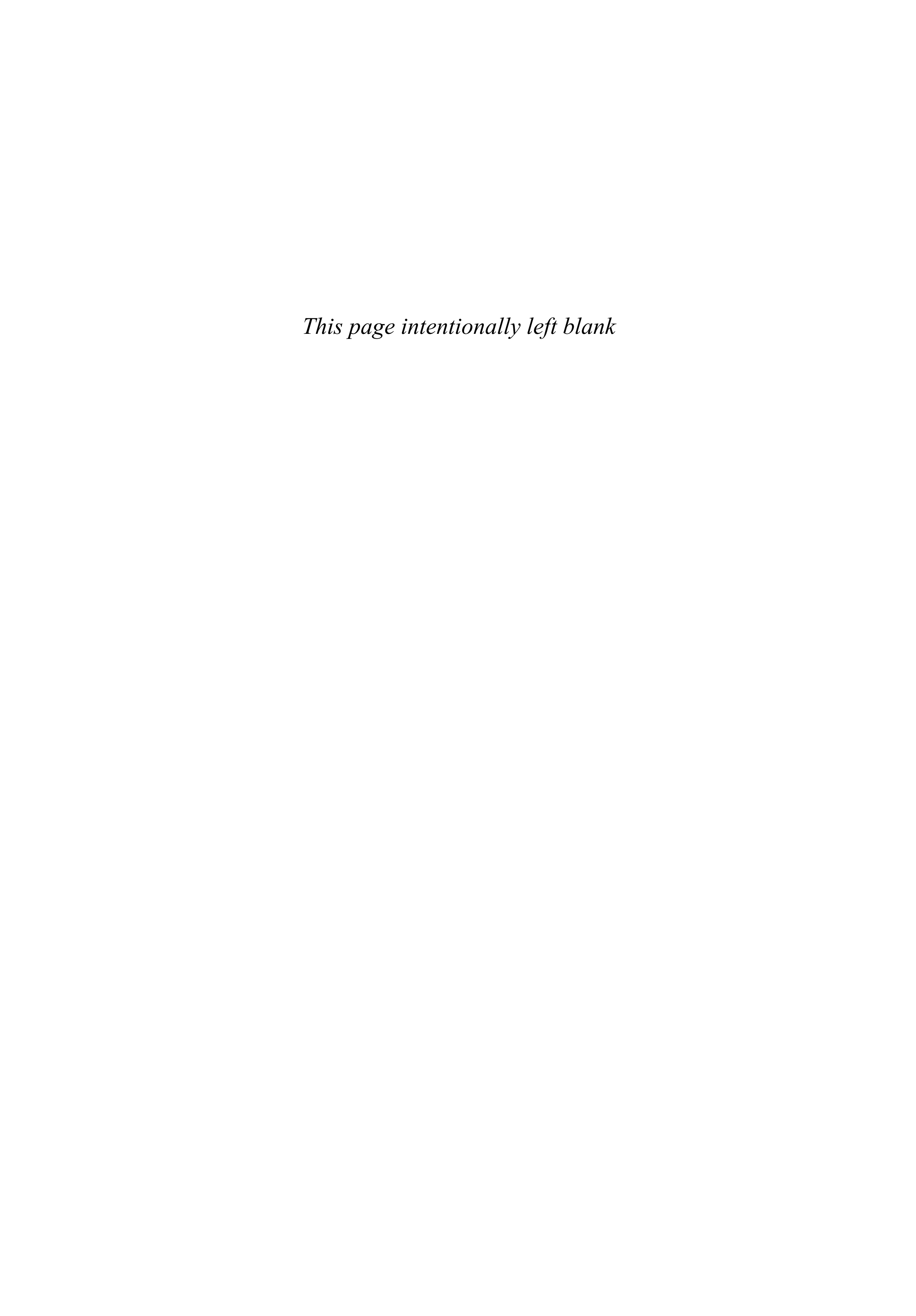
Usage

To run the script, follow these steps:

1. Select the script in the Navigator window.
2. Choose the created `GroovyShell` configuration from the Run menu (see Figure B.3).
3. Enter the arguments to be passed to the script.

You should be able to see the results in the console window.

FIGURE B.3 Running the script



This page intentionally left blank

Appendix C

Installing JSR 223

This appendix contains instructions for successfully installing and configuring the Scripting API reference implementation (JSR 223 RI). You can find more details on the specification in Chapters 9, “Scripting API,” and 10, “Web Scripting Framework.”

NOTE

Since the public review draft (which includes this reference implementation), there were some minor changes in the API, such as class and method name changes and explicit usage of Java SE 5 features. Examples in Chapter 9 are adjusted to these changes, so some of the examples might not work with this implementation. Please refer to the final version of the specification (http://jcp.org/aboutJava/communityprocess/pfd/jsr223/index.html) for more details.

Requirements

First, you need to download the Reference Implementation (RI) of the specification. You can do it by visiting http://jcp.org/aboutJava/communityprocess/pr/jsr223/index.html.

The RI is installed using the install script, which is written in Perl. So if you don't have the Perl interpreter (version 5.6.1) installed on your host, it's time to install it. You can find the Perl interpreter installation for various platforms at ActiveState's Web site, http://activestate.com/Products/ActivePerl/.

JSR 223 contains the Web Scripting Framework API specification. The RI of this API is targeted for the Apache Tomcat servlet container. If you want to use it and try out examples from Chapter 10, you have to have Tomcat installed before trying to set up JSR 223 RI. The JSR was developed using Tomcat version 5.0.19, so it is recommended that you use this version for running the examples. The RI should work with other 5.x family versions of Tomcat (and its successors), but this is not guaranteed. You can download the 5.0.19 version of Tomcat from http://archive.apache.org/dist/tomcat/tomcat-5/archive/v5.0.19/bin/.

Installation

As I already said, you install the RI using the Perl script. To start this process, just type the following in the directory that contains files from the RI's archive:

```
$ perl setup.pl
```

During installation, you are asked to enter locations for the JSR files and the Tomcat installation. The installer does everything else. The resulting output should be similar to the following:

```
Enter the jsr223 install location : [C:/jsr223] C:\dev\jsr223
Do you have tomcat installed  (y/n)? [n] y

Enter the installed Tomcat location  (): C:\dev\tomcat5

Installing JSR223 on C:\dev\jsr223

7 File(s) copied
0 File(s) copied
130 File(s) copied
1 File(s) copied

Installing PHP on C:\dev\jsr223/php5

7 File(s) copied

Configuring Tomcat

Stopping Tomcat
5 File(s) copied
Generated startupjsr.bat successfully
C:\dev\tomcat5/bin/startup.bat Modified
No change in Configuration file C:\dev\tomcat5/conf/web.xml
Modifying C:\dev\tomcat5/conf/server.xml
File C:\dev\tomcat5/conf/server.xml is backed up to
C:\dev\tomcat5/conf/server.xml.bak1
New configured tomcat will run only when started
using command line startup.bat
```

As you can see, the script changed Tomcat's configuration files, so you need to be sure that the configuration remained valid. If you encounter problems, you can always restore backup files and try again on the fresh installation of Tomcat 5.0.19.

After the install process is finished, you can test the installation:

```
Do you want to Test Tomcat for jsr223  (y/n)? [n]
Enter the tomcat port number : : [8080]
```

If no errors occurred during configuration testing, you should get output similar to this:

```
Testing Tomcat for localhost:8080

Stopping Tomcat
Starting Tomcat
Sleeping for 20 second to wait for Tomcat to start up


+++++++++ homepage : WORKED ++++++++++++++

+++++++++ php test page : WORKED ++++++++++++++

+++++++++ javascript test page : WORKED ++++++++++++++

Stopping Tomcat
Press Return to Continue :
Samples are copied in C:\dev\jsr223/samples directory

Installation Complete
```

As you can see, samples are copied to the `samples/` folder of the RI install folder.

INDEX

A

Accessing Application Context listing (10.2), 463
Active File Example listing (8.20), 376
Active File Generator listing (8.22), 380
active file pattern, 375
- consequences, 375
- problem, 375
- sample code, 376-380
- solution, 375

Active File Template listing (8.21), 379
ADD keyword, 5
addClassPath( ) method, 324
administration
- scripting, 328-334
- scripting languages, 55-58

Administration Script Example listing (7.17), 329
Advanced Ant BSF Support Example listing (7.13), 315
Advanced AntBuilder Example listing (7.15), 320
Advanced Binding Example—Java Application listing (9.13), 408
Advanced Groovy Programming Example listing (5.22), 225
AJAX (Asynchronous JavaScript And XML), 66
Ant build tools, 309-322
Ant BSF Support Example listing (7.12), 314
Ant task, Groovy, 132-133
Ant Task That Compiles All Scripts Inside the Project listing (8.10), 357
AntBuilder Example listing (7.14), 317
any( ) method, Groovy, 175-177
any( ) Method listing (4.30), 175
Apache Web servers, 62
- BSF (Bean Scripting Framework), 94

APIs (application programming interfaces), 49
- Java, 80-82

append( ) method, Groovy, 181-182
append( ) Method listing (4.39), 182
application scope, Web environments, 449
application variable (Groovlet), 215
applications
- BSF (Bean Scripting Framework), 275
  - *JSP (JavaServer Pages), 275-280*
  - *Xalan-J (XSLT), 280-287*
- Java, 79
- web applications, 59, 61-67
  - *ASP (Active Server Pages), 64*
  - *games, 68-69*
  - *JavaScript, 65-67*
  - *Perl, 61-62*
  - *PHP, 62-64*
  - *UNIX, 68*

apply( ) method, 263-264
architecture
BSF (Bean Scripting Framework), 248-249
compilers, 9
hybrid compiler-interpreters, 11
interpreters, 10
Java, 80-82
Scripting API, 391
Web Scripting Framework, 448, 482-488
context, 448-449
interaction, 451-453
servlet, 449-451
arguments, command-line arguments, Groovy, 136
Ascher, David, 29
ASP (Active Server Pages), 64
assembly languages, 5
assertArrayEquals( ) method, 298
assertContains( ) method, 299
assertEquals( ) method, 294
assertion methods, unit testing, 297-300
assertLength( ) method, 298
assertScript( ) method, 299
assertToString( ) method, 299
attributes, script context, 419-423
autoboxing
BeanShell, 89-90
Groovy, 140
Autoboxing listing (4.7), 140
AutoCAD, 50

B

basic syntax
BeanShell, 86-87
Python, 101-103
Bean Scripting Framework (BSF). *See* BSF **(Bean Scripting Framework)**
beans, BSF (Bean Scripting Framework)
declaring, 268-270
registering, 265-268
BeanShell, 80, 83-98, 126
commands, 91
downloading, 83
GUI (graphical user interface), 84
Java, embedding with, 94-98
manual, 98
methods, 91-92
modes, 84
objects, 92
running, 84
scripting
autoboxing, 89-90
basic syntax, 86-87
for loop, 88-89
interface implementation, 93-94
JavaBeans, 88
loosely typed syntax, 87
switch-case statement, 90
Workspace Editor, 85
"Best Tool for the Job, The", 36
Bezroukov, Nikolai, 68
binding, 186
dynamic binding, 442
programmatic binding, 442
Scripting API, 404-411
dynamic binding, 442-444
engine scope, 405-411
global scope, 411-416
script context, 416-428
Binding Example listing (9.11), 406
Binding listing (4.44), 186
bindings, Web Scripting Framework, 462
application, 462-464
request handling, 464-467
response, 468
servlet, 468-469
bootstrap class loaders (JVM), 81
Broadcasters Example listing (8.17), 367
broadcasters pattern, 359
consequences, 362
problem, 359-360
related patterns, 369

sample code, 362-369
solution, 360-362
BSF (Bean Scripting Framework), 94, 245-246
Ant scripting, 313-316
applications, 275
JSP (JavaServer Pages), 275-280
Xalan-J (XSLT), 280-287
architecture, 248-249
beans
declaring, 268-270
registering, 265-268
compilation, 270-275
data binding, 264-270
downloading, 247
engine initialization, 252
exception handling, 255
functions, 259-264
manager initialization, 252
methods, 259-264
origins of, 247
script files, executing, 257-259
Scripting API, compared, 445
scripting language registration, 249-252
scripts, 253-257
BSF Exception Handling listing (6.2), 256
BSF JSP Example listing (6.20), 279
BSFEngine.apply( ) Method Example listing (6.7), 263
BSFEngine.eval( ) Method Example listing (6.1), 254
BSFEngine.exec( ) Method Example listing (6.3), 256
BuilderSupport, Groovy, 235-236
building tools, Ant scripting, 309-322
built-in functions, 19
bytecode, 7

C

C++, Java, compared, 78
call( ) method, 169, 260
Cascading Style Sheets (CSS), 66
CBD (component-based development), 40
changed( ) method, 347
class files, Java, 80
class loaders (JVM), 81
bootstrap class loaders, 81
user-defined class loaders, 82
classes, 14
Groovy, 159-161
JavaAdapter class, 114
Classes listing (4.17), 159
classpaths, Groovy, setting, 131-132
closures
Groovy, 168-177
curly braces, 172
resource handling, 176-177
method closure, 92
methods, curly braces, 172
scripting language, 20
Closures listing (4.23), 170
code. *See also* listings; scripts
active file pattern, 376-380
as data, 19-23
bytecode, 7
development speed, 28-29
extension point pattern, 371-374
mediator scripting pattern, 345-354
observer pattern, 362-369
operation code, 4
runtime performance, 26-28
script object factory pattern, 356-359
scripted components pattern, 340-341
source code, production environment, 12-13
code generation, Scripting API, 428
method call syntax, 429-431
output statement, 429
programs, 431-432
collect( ) method, Groovy, 173
collect( ) Method listing (4.25), 173
collections, Groovy, 148
lists, 149-151
maps, 152-153
ranges, 151-152

command-line arguments, Groovy, 136
commands, 19
  BeanShell, 91
compatibility, Java and Groovy, 137
Compilable interface, 437-440
Compilable listing (9.35), 438
compilation, BSF (Bean Scripting Framework), 270-275
compilation languages, 52
Compile Example listing (6.13), 271
Compile Example—Result listing (6.14), 271
compile( ) method, 275
Compiled Script Usage Example listing (6.18), 274
compileExpr( ) method, 272
compilers, 6
  architecture, 9
  hybrid compiler-interpreters, 11
  interpreters, compared, 8-12
compiling scripts, Groovy, 130-133
Compiling Groovy Scripts with Ant listing (4.2), 132
component-based development (CBD), 40
components pattern (scripting), 337
  consequences, 339
  problem, 337-338
  related patterns, 341
  sample code, 340-341
  solution, 338-339
composite types, programming languages, 14
computer programs, 5
configuration
  Groovy, 492
  Web Scripting Framework, 456-462
consequences
  active file pattern, 375
  extension point pattern, 370
  mediator scripting pattern, 345
  observer pattern, 362
  script object factory pattern, 356
  scripted components pattern, 339
context, Web Scripting Framework architecture, 448-449
Creating an Sql Object with DataSource listing (5.6), 202
Creating Documents with DomBuilder listing (5.28), 233
cross-platform portability, Java, 77
CSS (Cascading Style Sheets), 66
curly braces, closures, methods, 172
currentThread( ) method, 444
Custom Application Context listing (9.26), 424
Custom BuilderSupport Implementation (BuilderSupport.java) listing (5.30), 235
Custom Engine Writer Example listing (9.28), 427
custom namespaces, 423-427
Customized compileExpr( ) Method listing (6.15), 272
Cygwin, 323

D

DAO (Database Access Object) design paradigm, 196
data binding, BSF (Bean Scripting Framework), 264-270
data structures, scripting languages, 17-19
Database Access Object (DAO) design paradigm, 196
database queries, GroovySQL, 202-205
databases, transactions, GroovySQL, 208-209
datasets, GroovySQL, 209-212
Davidson, James Duncan, 52, 311
debugging, interactive debugging, 304-309
Declare Bean Example—Java Application listing (6.11), 269
Declare Bean Example—Script listing (6.12), 270
declareBean( ) method, 268-269
declaring beans, BSF (Bean Scripting Framework), 268-270

Defining a Custom Namespace listing (9.25), 424
dependencies, Groovy, 131
design patterns, scripting, 335-337
  active file pattern, 375-380
  extension point pattern, 369-374
  mediator pattern, 341-354
  observer pattern, 359-369
  script object factory pattern, 354-359
  scripted components pattern, 337-341
Design Patterns, 336
Determining the Attribute's Scope listing (9.21), 420
development, CBD (component-based development), 40
development speed, code, 28-29
DHTML (Dynamic Hypertext Markup Language), 66
Dickerson, Chad, 56
disabling scripting, Web Scripting Framework, 456-457
discovery mechanism, Scripting API, 391-393
Discovery Mechanism listing (9.1), 393
DomBuilder, Groovy, 232-233
doSomeAction( ) method, 374
downloading
  BeanShell, 83
  BSF (Bean Scripting Framework), 247
  Groovy, 491
  Jython, 98
  Rhino, 110
dynamic binding, 442, Scripting API, 442-444
Dynamic Binding—Java Classes listing (9.39), 443
Dynamic Binding—Java Objects listing (9.38), 443
Dynamic Hypertext Markup Language (DHTML), 66
dynamic languages, 6
"Dynamic Languages—Ready for the Next Challenges, by Design," 29
dynamic linking, Java, 78
dynamic typing, 15-16
  Groovy, 139-140
Dynamic Typing and Polymorphism listing (4.6), 139

E

each( ) method, GroovySQL, 171-172, 211
each( ) Method listing (4.24), 172
each( ) Method listing (5.15), 211
eachByte( ) method, Groovy, 180-181
eachByte( ) Method listing (4.36), 181
eachFile( ) method, Groovy, 182
eachFile( ) Method listing (4.40), 182
eachFileRecurse( ) method, Groovy, 182
eachFileRecurse( ) Method listing (4.41), 182
eachLine( ) method, Groovy, 179
eachLine( ) Method listing (4.33), 179
eachRow( ) method, GroovySQL, 202-203
eachRow( ) Method and GString Query Parameters listing (5.7), 202
eachRow( ) Method and List Query Parameters listing (5.8), 203
Eckel, Bruce, 30
Eclipse IDE, 130
embedding
  BeanShell with Java, 94-98
  Groovy, 184-190
  Pyhton with Java, 108-109
  Rhino with Java, 114-120
  scripting languages, 70
engine initialization, BSF (Bean Scripting Framework), 252
engine interfaces, Scripting API, 432
  Compilable interface, 437-440
  Incovable interface, 432-437
engine metadata, Scripting API, 393-395
Engine Metadata listing (9.2), 394
engine readers, 427-428
engine scope
  binding, Scripting API, 405-411
  overriding, 407-409

engine writers, 427-428
eval( ) method, 19, 96, 400-404
evaluate( ) method, 184
evaluating scripts, Scripting API, 400-404
Evaluating a Script in the Script Context listing (9.20), 418
Evaluating Groovy Scripts from Java listing (4.43), 184
Evaluating Inline Scripts Under Security Policy listing (4.49), 193
Evaluating Script Contained in a File listing (9.8), 401
Evaluating Script Contained in a String listing (9.7), 400
every( ) method, Groovy, 175
every( ) Method listing (4.29), 175
evolutionary prototyping, 46
Example php.ini Configuration listing (9.37), 442
exception handling
- BSF (Bean Scripting Framework), 255
- Python, 107-108

exec( ) method, 270
execute( ) method, GroovySQL, 205-206
execute( ) Method listing (5.10), 205
executeUpdate( ) method, GroovySQL, 204-205
executeUpdate( ) Method and CSV Files listing (5.9), 204
executing script files, BSF (Bean Scripting Framework), 257, 259
Executing a Script File listing (6.4), 258
execution engine (JVM), 81-82
explicit memory management, 78
exportAsCSV( ) method, 377
exportAsSQL( ) method, 377
exportAsXML( ) method, 377
expressions, regular expressions, Groovy, 146-148
Extended XSLT Example listing (6.22), 282
Extending the switch-case Mechanism listing (4.15), 156
extensibility, scripting language, 70-71
Extensible Stylesheet Language Transformation (XSLT), 280
Extension Point Example listing (8.18), 371
extension point pattern, 369
- consequences, 370
- problem, 369
- related patterns, 374
- sample code, 371-374
- solution, 370

extreme programming, 33-35

F

fail( ) method, 294
files, system operations, Groovy, 178-182
filter( ) function, 21
filter( ) method, 22
filters, 41
find( ) method, Groovy, 174
find( ) Method listing (4.27), 174
findAll( ) method, Groovy, 174-175
findAll( ) method, GroovySQL, 211-212
findAll( ) Method listing (4.28), 174
findAll( ) Method listing (5.16), 212
Foemmel, Matthew, 292
Font dialog box, 351
for loop
- BeanShell, 88-89
- Groovy, 157

for Loop listing (4.16), 157
forward( ) method, Web Scripting Framework, 471-473
Fowler, Martin, 292
frame( ) method, 237
Function Call Example listing (6.5), 260
functions
- as method arguments, 21-23
- BSF (Bean Scripting Framework), 259-264
- built-in functions, 19
- eval( ), 19
- filter( ), 21

Incovable interface, 433-434
over( ), 21
printColor( ), 140
someFunc( ), 103
upper( ), 102

G

games, scripting in, 68-69
garbage collector (JVM), 78
General Scripting API, 389
generating code, Scripting API, 428-432
generics, Java, 31
get( ) method, 95
getBindings( ) method, 412
getComponent( ) method, 347
getEngineByExtension( ) method, 398
getEngineByExtension( ) Method Example listing (9.5), 398
getEngineByMimeType( ) method, 398
getEngineByName( ) method, 397
getEngineByName( ) Method Example listing (9.4), 397
getInterface( ) method, 97
getLanguageName( ) method, 394
getMethodCallSyntax( ) method, 429-431
getOutputStatement( ) method, 429
getScriptEngine( ) method, 395-396
getScriptEngine( ) Method Example listing (9.3), 395
getter/setter methods, 165
getText( ) Method listing (4.32), 178
getText( ) method, Groovy, 178
global scope
binding, Scripting API, 411-416
initialization, 415-416
variables, overriding, 413-414
Global Scope Example listing (9.15), 412
Global Scope Initialization listing (9.18), 415
Glue Code Client listing (8.8), 352
glue code pattern, 341
consequences, 345
problem, 341-342
related patterns, 354
sample code, 345-354
solution, 342-345
glue languages, 41
graphical user interfaces (GUIs). *See* GUIs (graphical user interfaces)
Groovlet Deployment Descriptor listing (5.17), 214
Groovlet Example listing (5.18), 218
Groovlets, 212-219
variables, 214
Groovy programming language, 80, 120-121, 125, 194-195
advantages of, 126
Ant task, 132-133
any( ) method, 175-177
autoboxing, 140
BuilderSupport, 235-236
classes, 159-161
classpaths, setting, 131-132
closures, 168-177
curly braces, 172
resource handling, 176-177
collect( ) method, 173
collections, 148
lists, 149-151
maps, 152-153
ranges, 151-152
configuring, 492
DomBuilder, 232-233
downloading, 491
dynamic typing, 139-140
each( ) method, 171-172
every( ) method, 175
find( ) method, 174
findAll( ) method, 174-175
Groovlets, 212-219
variables, 214
GroovyBeans, 165
named parameters, 166-167
object navigation, 167
properties, 165-166
safe navigation, 168

GroovySQL, 196-212
  database queries, 202-205
  datasets, 209-212
  each( ) method, 211
  eachRow( ) method, 202-203
  execute( ) method, 205-206
  executeUpdate( ) method, 204-205
  findAll( ) method, 211-212
  object creation, 199-207
  prepared statements, 206-207
  Sql.loadDriver( ) method, 199-200
  stored procedures, 207-208
  transactions, 208-209
IDE support, 495-497
inject( ) method, 173-174
installing, 127, 491-492
interactive console, 128-129
interactive shell, 127-128
JAR (Java Archive) files, 491
Java
  compatibility with, 137
  embedding with, 184-190
  statements, 138
language syntax, 137-177
  triple-quote syntax, 144-145
logical branching, 154-156
looping, 156-158
loose typing, 138-140
Markup syntax, 223-236
NamespaceBuilder, 234
NodeBuilder, 227-229
operator overloading, 162-164
polymorphism, 139-140
regular expressions, 146-148
SaxBuilder, 230-231
script files, evaluating, 129-130
scripts
  command-line arguments, 136
  compiling, 130-133
  dependencies, 131
  running, 127-130
  structure, 133, 135-136
security, 190-194
strings, 143-145
  GStrings, 145-146
Swing user interfaces, 236-243
switch-case structure, 154-156
system operations, 178
  files, 178-182
  processes, 182-183
templates, 220-223
testing, 492-493
type juggling, 140-143
Groovy Script Structure listing (4.3), 134
Groovy Shell listing (4.1), 128
Groovy Template Support listing (5.19), 221
GroovyBeans, 165
  named parameters, 166-167
  object navigation, 167
  properties, 165-166
  safe navigation, 168
GroovyBeans listing (4.21), 165
GroovySQL, 196-212
  datasets, 209-212
  each( ) method, 211
  eachRow( ) method, 202-203
  execute( ) method, 205-206
  executeUpdate( ) method, 204-205
  findAll( ) method, 211-212
  objects, creation, 199-207
  prepared statements, 206-207
  queries, database queries, 202-205
  Sql.loadDriver( ) method, 199-200
  stored procedures, 207-208
  transactions, 208-209
GroovySQL Example listing (5.1), 197
GroovyTestCase class, unit testing, 296-300
GroovyTestCase Example listing (7.2), 296
GroovyTestCase.assertArrayEquals( ) Method Example listing (7.4), 298
GroovyTestCase.assertContains( ) Method Example listing (7.5), 299

GroovyTestCase.assertLength( ) Method Example listing (7.3), 298
GroovyTestCase.assertScript( ) Method Example listing (7.7), 299
GroovyTestCase.assertToString( ) Method Example listing (7.6), 299
GroovyTestCase.shouldFail( ) Method Example listing (7.8), 300
GroovyTestSuite Example listing (7.10), 301
GStrings, Groovy, 145-146
GStrings listing (4.9), 145
GUIs (graphical user interfaces), 58
    BeanShell, 84

H

handling
    command-line arguments, Groovy, 136
    script evaluation results, 402
    ScriptException, 403-404
    sessions, Web Scripting Framework, 473-478
Handling Command-Line Arguments in Groovy listing (4.4), 136
Handling Script Evaluation Result listing (9.9), 402
Handling ScriptException listing (9.10), 403
high-level languages, 6
HTML (Hypertext Markup Language), 60
    CSS (Cascading Style Sheets), 66
    DHTML (Dynamic Hypertext Markup Language), 66
HTTP (HyperText Transfer Protocol), 60
HTTP request handling, Web Scripting Framework, 464-467
HTTP Request Handling—HTML Form listing (10.3), 465
HTTP Request Handling—PHP Script listing (10.4), 466
Hunt, Andy, 53
hybrid compiler-interpreters, 11
hypertext, 60

I

IDEs, 130
    Groovy, support for, 495-497
Implementing Java Interfaces in Groovy listing (4.46), 189
implicit memory management, 78
include( ) method, Web Scripting Framework, 469-471
Including Resources listing (10.5), 470
Incovable interface, 432
    functions, 433-434
    interfaces, 435-437
    methods, 434-437
initialization
    engines, BSF (Bean Scripting Framework), 252
    global scope, 415-416
    managers, BSF (Bean Scripting Framework), 252
    Sql object with connection, 200-201
Initializing the Sql Object with a Connection—Groovy Script listing (5.5), 201
Initializing the Sql Object with a Connection—Java Class listing (5.4), 200
inject( ) method, Groovy, 173-174
inject( ) Method listing (4.26), 173
installation
    Groovy, 127, 491-492
    JSR 223 RI (Reference Implementation), 499-502
    Jython, 99
interaction, Web Scripting Framework architecture, 451-453
interactive console (Groovy), scripts, running, 128-129
interactive debugging, 304-309
Interactive Debugging with the Groovy Shell listing (7.11), 306
interactive mode, BeanShell, 84
interactive shell (Groovy), scripts, running, 127-128
Intercepting Method Calls listing (4.18), 161

interfaces
- Compilable interface, 437-440
- engine interfaces, Scripting API, 432-440
- implementing
  - *BeanShell, 93-94*
  - *Python, 105-107*
  - *Rhino, 112-114*
- Incovable interface, 435-437

Interfaces listing (9.34), 436
interpreters, 6
- architecture, 10
- compilers, compared, 8-12
- hybrid interpreter-compilers, 11

Introductory DataSet Example listing (5.14), 209
Introductory JSP Example listing (6.19), 277
Introductory MarkupBuilder Example listing (5.21), 224
Invocable Example listing (9.32), 433
invoice( ) method, 92
invokeFunction( ) method, 434
isCase( ) method, 155-156

J

Jacl, 122
Jakarta Commons Logging project, 247
JAR (Java Archive) files, Groovy, 491
Java
- API (application programming interface), 80-82
- applications
  - *planning, 79*
  - *speed issues, 79*
- architecture, 80-82
- BeanShell, embedding with, 94-98
- C++, compared, 78
- class files, 80
- commercial release of, 77
- cross-platform portability, 77
- dynamic linking, 78
- generics, 31
- Groovy
  - *compatibility with, 137*
  - *embedding with, 184-190*
- implicit memory management, 78
- JVM (Java Virtual Machine), 78-82
  - *BeanShell, 83-98*
  - *class loaders, 81-82*
  - *execution engine, 81-82*
  - *garbage collector, 78*
  - *Jython, 98-109*
  - *other programming languages, 82-83*
  - *Python, 98-109*
  - *Rhino, 110-120*
- popularity of, 79
- programming
  - *administration, 328-334*
  - *Ant scripting, 309-322*
  - *interactive debugging, 304-309*
  - *management, 328-334*
  - *shell scripting, 323-328*
  - *unit testing, 292-304*
- programming language, 80
- Python
  - *embedding with, 108-109*
  - *scripting in, 103-105*
- Rhino
  - *embedding with, 114-120*
  - *scripting in, 111-112*
- single inheritance model, 78
- treading, 77

Java Community Process (JCP), 121, 385
Java Database Connectivity (JDBC) driver, 196
Java Runtime Environment (JRE), 24
Java Specification Request (JSR), 67
Java Specification Request 241, 121
Java Virtual Machine (JVM). *See* **JVM (Java Virtual Machine)**
JavaAdapter class, Rhino, 114
JavaBeans, BeanShell, 88
Javagator project, Netscape Navigator, 110

JavaScript, web applications, 65-67
JCP (Java Community Process), 121, 385
JDBC (Java Database Connectivity) driver, 196
Jetty servlet, 213
JRE (Java Runtime Environment), 24
JRuby, 122
JSP (Java Server Pages), BSF (Bean Scripting Framework), 275-280
JSR (Java Specification Request), 67
JSR 223 RI (Reference Implementation), installing, 499-502
JudoScript, 122
JUnit, 293-295
JUnit Example listing (7.1), 294
JVM (Java Virtual Machine), 7, 24-25, 78-82
    BeanShell, scripting in, 83-98
    class loaders, 81
        bootstrap class loaders, 81
        user-defined class loaders, 82
    execution engine, 81-82
    garbage collector, 78
    Jython, scripting in, 98-109
    other programming languages, using in, 82-83
    Python, scripting in, 98-109
    Rhino, scripting in, 110-120
Jython, 80, 98-109, 126
    downloading, 98
    installing, 99
    switches, 101

K-L

keys, reversed keys, 410-411
keywords (ADD), 5

Label Widget listing (8.4), 347
LAMP (Linux, Apache, MySQL, PHP), 386
language syntax, Groovy, 137-177
    triple-quote syntax, 144-145
language tags, Web Scripting Framework, 478-481
Language Tags—Configuration listing (10.11), 480
Language Tags—Groovy Script listing (10.12), 481
Language Tags—Original Groovy Script listing (10.10), 479
languages
    compilation languages, 52
    programming languages
        assembly languages, 5
        code as data, 19-23
        compilers, 6
        data structures, 17-19
        Groovy programming language, 80, 120-121
        high-level languages, 6
        JRuby, 122
        JudoScript, 122
        ObjectScript, 123
        OOP (object-oriented programming) languages, 40
        third-generation programming languages, 6
        typing, 13-17
    scripting languages, 4-24
        administration, 55-58
        closures, 20
        compilers, 8-12
        customization, 49-51
        embedding, 70
        extensiblity, 70-71
        Groovy, 194
        interpreters, 8-12
        Java, 80-82
        learnability, 71-72
        management, 55-58
        method arguments, 21-23
        Perl, 43
        prototyping models, 44-48
        Python, 18
        software development support, 51-55

source code, 12-13
Tcl (Tool Command Language), 37-44
Tcl/Java, 122
UNIX shell languages, 41-42
user interface programming, 58-59
virtual machines, 24-25
web applications, 59-69
wiring, 40-44
Web Scripting Framework, allowing, 459
Linux, Apache, MySQL, PHP (LAMP), 386
List Box Widget listing (8.5), 348
listings
4.1 (Groovy Shell), 128
4.2 (Compiling Groovy Scripts with Ant), 132
4.3 (Groovy Script Structure), 134
4.4 (Handling Command-Line Arguments in Groovy), 136
4.5 (Loose Typing), 138
4.6 (Dynamic Typing and Polymorphism), 139
4.7 (Autoboxing), 140
4.8 (Triple-Quote Syntax), 144
4.9 (GStrings), 145
4.10 (Regular Expressions), 147
4.11 (Lists), 150
4.12 (Ranges), 151
4.13 (Maps), 153
4.14 (switch-case Structure), 154
4.15 (Extending the switch-case Mechanism), 156
4.16 (for Loop), 157
4.17 (Classes), 159
4.18 (Intercepting Method Calls), 161
4.19 (Operators), 163
4.20 (Operator Overloading), 164
4.21 (GroovyBeans), 165
4.22 (Safe Navigation), 168
4.23 (Closures), 170
4.24 {each( ) Method}, 172
4.25 {collect( ) Method}, 173
4.26 {inject( ) Method}, 173
4.27 {find( ) Method}, 174
4.28 {findAll( ) Method}, 174
4.29 {every( ) Method}, 175
4.30 {any( ) Method}, 175
4.31 (Resource Handling with Closures), 176
4.32 {getText( ) Method}, 178
4.33 {eachLine( ) Method}, 179
4.34 {readLines( ) Method}, 179
4.35 {splitEachLine( ) Method}, 180
4.36 {eachByte( ) Method}, 181
4.37 {readBytes( ) Method}, 181
4.38 {write( ) Method}, 181
4.39 {append( ) Method}, 182
4.40 {eachFile( ) Method}, 182
4.41 {eachFileRecurse( ) Method}, 182
4.42 (Process Handling), 183
4.43 (Evaluating Groovy Scripts from Java), 184
4.44 (Binding), 186
4.45 (Loading Groovy Scripts as Java Classes), 187
4.46 (Implementing Java Interfaces in Groovy), 189
4.47 (Security Example), 191
4.48 (Security Policy), 191
4.49 (Evaluating Inline Scripts Under Security Policy), 193
5.1 (GroovySQL Example), 197
5.2 (SQL Object Creation Alternatives), 199
5.3 {Sql.loadDriver( ) Method}, 199
5.4 (Initializing the Sql Object with a Connection—Java Class), 200
5.5 (Initializing the Sql Object with a Connection—Groovy Script), 201
5.6 (Creating an Sql Object with DataSource), 202
5.7 {eachRow( ) Method and GString Query Parameters}, 202

5.8 {eachRow( ) Method and List Query Parameters}, 203
5.9 {executeUpdate( ) Method and CSV Files}, 204
5.10 {execute( ) Method}, 205
5.11 (Stored Procedure), 207
5.12 (Stored Procedure Call), 208
5.13 (Transactions), 208
5.14 (Introductory DataSet Example), 209
5.15 {each( ) Method}, 211
5.16 {findAll( ) Method}, 212
5.17 (Groovlet Deployment Descriptor), 214
5.18 (Groovlet Example), 218
5.19 (Groovy Template Support), 221
5.20 (Template Example), 222
5.21 (Introductory MarkupBuilder Example), 224
5.22 (Advanced Groovy Programming Example), 225
5.23 (XHTML Markup Example), 226
5.24 (NodeBuilder Example), 227
5.25 (SAX Handler Example), 230
5.26 (SaxBuilder), 231
5.27 (Parsing Documents with DomBuilder), 232
5.28 (Creating Documents with DomBuilder), 233
5.29 (Namespace Example), 234
5.30 {Custom BuilderSupport Implementation (BuilderSupport.java)}, 235
5.31 (SwingBuilder), 237
5.32 (TableLayout), 239
5.33 (TableModel), 241
6.1 {BSFEngine.eval( ) Method Example}, 254
6.2 (BSF Exception Handling), 256
6.3 {BSFEngine.exec( ) Method Example}, 256
6.4 (Executing a Script File), 258
6.5 (Function Call Example), 260
6.6 (Method Call Example), 262
6.7 {BSFEngine.apply( ) Method Example}, 263
6.8 (Register Bean Example—JavaBean Definition), 266
6.9 (Register Bean Example—Java Application), 267
6.10 (Register Bean Example—Script), 267
6.11 (Declare Bean Example—Java Application), 269
6.12 (Declare Bean Example—Script), 270
6.13 (Compile Example), 271
6.14 (Compile Example—Result), 271
6.15 {Customized compileExpr( ) Method}, 272
6.16 (Modified Compile Example), 273
6.17 (Modified Compile Example—Result), 274
6.18 (Compiled Script Usage Example), 274
6.19 (Introductory JSP Example), 277
6.20 (BSF JSP Example), 279
6.21 (Simple XSLT Transformation), 281
6.22 (Extended XSLT Example), 282
6.23 (Xalan-J Extension Example), 283
6.24 (Xalan-J BSF Example), 285
7.1 (JUnit Example), 294
7.2 (GroovyTestCase Example), 296
7.3 {GroovyTestCase.assertLength( ) Method Example}, 298
7.4 (GroovyTestCase.assertArrayEquals( ) Method Example), 298
7.5 {GroovyTestCase.assertContains( ) Method Example}, 299
7.6 {GroovyTestCase.assertToString( ) Method Example}, 299
7.7 {GroovyTestCase.assertScript( ) Method Example}, 299
7.8 {GroovyTestCase.shouldFail( ) Method Example}, 300

7.9 (TestSuite Example), 300
7.10 (GroovyTestSuite Example), 301
7.11 (Interactive Debugging with the Groovy Shell), 306
7.12 (Ant BSF Support Example), 314
7.13 (Advanced Ant BSF Support Example), 315
7.14 (AntBuilder Example), 317
7.15 (Advanced AntBuilder Example), 320
7.16 (Shell Scripting Example), 327
7.17 (Administration Script Example), 329
7.18 (Script Job Example), 330
7.19 (Quartz Scheduler Configuration Example), 332
8.1 (Scripted Component Pattern Example), 340
8.2 (Mediator Interface), 346
8.3 (Mediator-Aware Widget Abstraction), 346
8.4 (Label Widget), 347
8.5 (List Box Widget), 348
8.6 (Mediator Implementation), 349
8.7 (Mediator Client), 350
8.8 (Glue Code Client), 352
8.9 (Modified Mediator Component), 353
8.10 (Ant Task That Compiles All Scripts Inside the Project), 357
8.11 (Scripted Object Factory Pattern Example), 357
8.12 (Subject Component), 363
8.13 (Observer Label Widget), 364
8.14 (Price Observer Label Widget), 364
8.15 (Tax Observer Label Widget), 365
8.16 (Observer Client), 365
8.17 (Broadcasters Example), 367
8.18 (Extension Point Example), 371
8.19 (Modified Extension Point Example), 372
8.20 (Active File Example), 376
8.21 (Active File Template), 379
8.22 (Active File Generator), 380
9.1 (Discovery Mechanism), 393
9.2 (Engine Metadata), 394
9.3 {getScriptEngine( ) Method Example}, 395
9.4 {getEngineByName( ) Method Example}, 397
9.5 {getEngineByExtension( ) Method Example}, 398
9.6 (Register Engine Example), 399
9.7 (Evaluating Script Contained in a String), 400
9.8 (Evaluating Script Contained in a File), 401
9.9 (Handling Script Evaluation Result), 402
9.10 (Handling ScriptException), 403
9.11 (Binding Example), 406
9.12 (Overriding the Engine Scope), 407
9.13 (Advanced Binding Example—Java Application), 408
9.14 (Reserved Keys), 411
9.15 (Global Scope Example), 412
9.16 (Variables Overriding—An Example), 413
9.17 (ScriptEngineManager's Shortcut Methods), 414
9.18 (Global Scope Initialization), 415
9.19 (Namespace Example), 416
9.20 (Evaluating a Script in the Script Context), 418
9.21 (Determining the Attribute's Scope), 420
9.22 (Obtaining an Attribute from the Desired Scope), 421
9.23 (Modifying Attributes in script—Script), 422
9.24 (Modifying Attributes in script—Java Application), 422
9.25 (Defining a Custom Namespace), 424

9.26 (Custom Application Context), 424
9.27 (Using Custom Application Context), 426
9.28 (Custom Engine Writer Example), 427
9.29 (Output Statement), 429
9.30 (Method Call Syntax), 430
9.31 (Program Example), 431
9.32 (Invocable Example), 433
9.33 (Method Call), 435
9.34 (Interfaces), 436
9.35 (Compilable), 438
9.36 (Threading), 441
9.37 (Example php.ini Configuration), 442
9.38 (Dynamic Binding—Java Objects), 443
9.39 (Dynamic Binding—Java Classes), 443
10.1 (Sample Web Application Descriptor), 454
10.2 (Accessing Application Context), 463
10.3 (HTTP Request Handling—HTML Form), 465
10.4 (HTTP Request Handling—PHP Script), 466
10.5 (Including Resources), 470
10.6 (Request Forwarding), 471
10.7 {Session Handling—Login Form (form.jsp)}, 474
10.8 (Session Handling—Login Servlet), 475
10.9 (Session Handling—PHP Script), 477
10.10 (Language Tags—Original Groovy Script), 479
10.11 (Language Tags—Configuration), 480
10.12 (Language Tags—Groovy Script), 481
10.13 (PHP to Java—PHP Class), 484
10.14 (PHP to Java—Factory Class), 485
10.15 (PHP to Java—Client Code), 486
10.16 (PHP to Java—Java Class), 486
10.17 (PHP to Java—Modified Factory Class), 487
lists, Groovy, 149-151
Lists listing (4.11), 150
LiveScript, 65
Loading Groovy Scripts as Java Classes listing (4.45), 187
logical branching, Groovy, 154-156
lookupBean( ) method, 266
looping, Groovy, 156-158
loops, for loop, BeanShell, 88-89
loose methods, 134
loose typing, Groovy, 138-140
Loose Typing listing (4.5), 138
loosely defined methods, 91
loosely typed syntax, BeanShell, 87
Loui, Ronald, 71
LucasArts, 69

M

machine instructions, 4-5
machine language, 5
main( ) method, 134, 295, 304
Make tool, 51
make( ) method, 222
Makefiles, 52
management, scripting, 55-58, 328-334
manager initialization, BSF (Bean Scripting Framework), 252
manuals, BeanShell, 98
maps, Groovy, 152-153
Maps listing (4.13), 153
Markup syntax, Groovy, 223-236
McIlroy, Doug, 40
Mediator Client listing (8.7), 350
Mediator Implementation listing (8.6), 349
Mediator Interface listing (8.2), 346
mediator scripting pattern, 341
  consequences, 345
  problem, 341-342

related patterns, 354
sample code, 345-354
solution, 342-345
Mediator-Aware Widget Abstraction listing (8.3), 346
metadata, engine metadata, Scripting API, 393, 395
method arguments, functions as, 21, 23
Method Call Example listing (6.6), 262
Method Call listing (9.33), 435
Method Call Syntax listing (9.30), 430
method closure, 92
method code syntax, Scripting API, 429-431
method overloading, 139
methods
addClassPath( ), 324
any( ) method, 175-177
append( ), 181
apply( ), 263-264
assertArrayEquals( ), 298
assertContains( ), 299
assertEquals( ), 294
assertLength( ), 298
assertScript( ), 299
assertToString( ), 299
BeanShell, 91-92
BSF (Bean Scripting Framework), 259-264
call( ), 169, 260
changed( ), 347
closures, curly braces, 172
collect( ) method, 173
compile( ), 275
compileExpr( ), 272
currentThread( ), 444
declareBean( ), 268-269
doSomeAction( ), 374
each( ), 211
each( ) method, 171-172
eachByte( ), 180
eachFile( ), 182
eachFileRecurse( ), 182
eachLine( ), 179
eachRow( ), 202-203
eval( ), 96, 400-404
evaluate( ), 184
every( ) method, 175
exec( ), 270
execute( ), 205-206
executeUpdate( ), 204-205
exportAsCSV( ), 377
exportAsSQL( ), 377
exportAsXML( ), 377
fail( ), 294
filter( ), 22
find( ) method, 174
findAll( ), 211-212
findAll( ) method, 174-175
forward( ) method, 471-473
frame( ), 237
get( ), 95
getBindings( ), 412
getComponent( ), 347
getEngineByExtension( ), 398
getEngineByMimeType( ), 398
getEngineByName( ), 397
getInterface( ), 97
getLanguageName( ), 394
getMethodCallSyntax( ), 429-431
getOutputStatement( ), 429
getScriptEngine( ), 395-396
getter/setter methods, 165
getText( ), 178
include( ) method, 469-471
Incovable interface, 434-437
inject( ) method, 173-174
invoice( ), 92
invokeFunction( ), 434
isCase( ), 155-156
lookupBean( ), 266
loosely defined methods, 91
main( ), 134, 295, 304
make( ), 222

method closure, 92
method overloading, 139
notifyObservers( ), 363
panel( ), 237
put( ), 406
readBytes( ), 181
readLines( ), 179
registerBean( ), 266, 268
registerEngineExtension( ), 399
registerEngineMimeType( ), 399
registerEngineName( ), 399
run( ), 134
script( ), 95
select( ), 22
sendError( ), 468
set( ), 95
setBindings( ), 407
setChanged( ), 363
setClassPath( ), 324
setEmail( ), 308
setPrice( ), 363
setUp( ), 294
shouldFail( ), 299
show( ), 350
showDialog( ), 346, 350
sleep( ), 444
source( ), 95-96
splitEachLine( ), 180
Sql.loadDriver( ), 199-200
standalone methods, 134
termination( ), 134
testAnimal( ), 156
testInit( ), 294
testIteration( ), 294
unregisterBean( ), 266
update( ), 238, 364
valueChanged( ), 348
waitFor( ), 183
widgetChanged( ), 346, 348, 350
write( ), 181
modes, BeanShell, 84
Modified Compile Example listing (6.16), 273
Modified Compile Example—Result listing (6.17), 274
Modified Extension Point Example listing (8.19), 372
Modified Mediator Component listing (8.9), 353
Modifying Attributes in script—Java Application listing (9.24), 422
Modifying Attributes in script—Script listing (9.23), 422
mod_perl, 62
***Mythical Man-Month: Essays on Software Engineering, The*, 48**

N

named parameters, GroovyBeans, 166-167
Namespace Example listing (5.29), 234
Namespace Example listing (9.19), 416
NamespaceBuilder, Groovy, 234
Namespaces, 234
custom namespaces, 423-427
script context, 416-419
native types, programming languages, 14
navigation
objects, GroovyBeans, 167
save navigation, GroovyBeans, 168
Netscape Navigator, Javagator project, 110
NodeBuilder, Groovy, 227-229
NodeBuilder Example listing (5.24), 227
notifyObservers( ) method, 363

O

O'Reilly, Tim, 61
object-oriented programming (OOP) languages, 40
object-oriented technology (OOT), 40
Object-Relational Mapping (ORM) tools, 196
objects
BeanShell, 92
creating, GroovySQL, 199-207
GroovyBeans, navigation, 167
GString objects, Groovy, 145-146

ObjectScript, 123
Observer Client listing (8.16), 365
Observer Label Widget listing (8.13), 364
observer pattern, 359
  consequences, 362
  problem, 359-360
  related patterns, 369
  sample code, 362-369
  solution, 360-362
Obtaining an Attribute from the Desired Scope listing (9.22), 421
OOP (object-oriented programming) language, 40
OOT (object-oriented technology), 40
operators, overloading, Groovy, 162-164
operands, 4
operartion code, 4
Operator Overloading listing (4.20), 164
Operators listing (4.19), 163
ORM (Object-Relational Mapping) tools, 196
Ousterhout, John K., 37
out variable (Groovlet), 214
Output Statement listing (9.29), 429
output statements, Scripting API, 429
over( ) function, 21
overloading operators, Groovy, 162-164
overloading methods, 139
overriding
  engine scope, 407-409
  variables, global scope, 413-414
Overriding the Engine Scope listing (9.12), 407

P

panel( ) method, 237
parameters, named parameters, GroovyBeans, 166-167
Parsing Documents with DomBuilder listing (5.27), 232
patterns, scripting, 335-337
  active file pattern, 375-380
  extension point pattern, 369-374
  mediator pattern, 341-354
  observer pattern, 359-369
  script object factory pattern, 354-359
  scripted components pattern, 337-341
Perl programming language, 43
  web applications, 61-62
Pettichord, Bret, 55
PHP, 386
  classes, 484
  client code, 486
  factory class, 486-487
  web applications, 62-64
PHP to Java—Client Code listing (10.15), 486
PHP to Java—Factory Class listing (10.14), 485
PHP to Java—Java Class listing (10.16), 486
PHP to Java—Modified Factory Class listing (10.17), 487
PHP to Java—PHP Class listing (10.13), 484
polymorphism, 139
  Groovy, 139-140
Portraits of Open Source Pioneers, 68
Pragmatic Unit Testing in Java with JUnit, 53
prepared statements, GroovySQL, 206-207
Price Observer Label Widget listing (8.14), 364
primitive types, programming languages, 14
print( ) command (BeanShell), 91
printColor( ) function, 140
problems
  active file pattern, 375
  extension point pattern, 369
  mediator scripting pattern, 341-342
  observer pattern, 359-360
  script object factory pattern, 355
  scripted components pattern, 337-338
procedures, stored procedures, GroovySQL, 207-208

Process Handling listing (4.42), 183
processes, system operations, Groovy, 182–183
processors
machine instructions, execution of, 4
machine language, 5
production environment, source code, 12–13
Program Example listing (9.31), 431
programmatic binding, 442
programming. *See also* scripting
extreme programming, 33-35
Java
administration, 328-334
Ant scripting, 309-322
interactive debugging, 304-309
management, 328-334
shell scripting, 323-328
unit testing, 292-304
UNIX, 40
user interface programming, 58-59
programming languages. *See also* scripting languages
assembly languages, 5
BeanShell, 83-98, 126
compilers, 6
Groovy, 80, 120-121, 125, 194-195
advantages of, 126
configuring, 492
downloading, 491
Groovlets, 212-219
GroovySQL, 196-212
IDE support, 495-497
installing, 127, 491-492
Markup syntax, 223-236
Swing user interfaces, 236-243
templates, 220-223
testing, 492-493
high-level languages, 6
Java, 80
JRuby, 122
JudoScript, 122
JVM (Java Virtual Machine), using in, 82-83
Jython, 98-109, 126
ObjectScript, 123
OOP (object-oriented programming) languages, 40
Python, 98-109, 126
Rhino, 110-114, 116-120
second-generation programming languages, 5
Tcl/Java, 122
third-generation programming languages, 6
typing, 13, 15
composite types, 14
dynamic typing, 15-16
native types, 14
user-defined types, 14
weak typing, 17
programs
computer programs, 5
Scripting API, 431-432
project building, scripting languages, 51–53
properties, GroovyBeans, 165–166
prototyping models, 44–48
evolutionary prototyping, 46
Python, 47-48
throwaway prototyping, 45-46
put( ) method, 406
Python scripting language, 18, 47–48, 98–109, 126
Java, embedding with, 108-109
scripting
basic syntax, 101-103
exception handling, 107-108
in Java, 103-105
interface implementation, 105-107

Q-R

Quartz scheduler, 330–334
Quartz Scheduler Configuration Example listing (7.19), 332

queries, database queries, GroovySQL, 202-205

ranges, Groovy, 151-152
Ranges listing (4.12), 151
readBytes( ) method, Groovy, 181
readBytes( ) Method listing (4.37), 181
readers, 427-428
readLines( ) method, Groovy, 179
readLines( ) Method listing (4.34), 179
refactoring, 33
Register Bean Example—Java Application listing (6.9), 267
Register Bean Example—JavaBean Definition listing (6.8), 266
Register Bean Example—Script listing (6.10), 267
Register Engine Example listing (9.6), 399
registerBean( ) method, 266, 268
registerEngineExtension( ) method, 399
registerEngineMimeType( ) method, 399
registerEngineName( ) method, 399
registering
  beans, BSF (Bean Scripting Framework), 265-268
  scripting engines, Scripting API, 399
  scripting languages, BSF (Bean Scripting Framework), 249-252
regular expressions (regex), Groovy, 146-148
Regular Expressions listing (4.10), 147
request forwarding, Web Scripting Framework, 471-473
Request Forwarding listing (10.6), 471
request handling, Web Scripting Framework, 464-467
request scope, Web environments, 448
request variable (Groovlet), 214
Reserved Keys listing (9.14), 411
resource handling, closures, Groovy, 176-177
Resource Handling with Closures listing (4.31), 176
responses, Web Scripting Framework, 468
results, Web Scripting Framework, displaying, 460-462
reversed keys, 410-411
Rhino, 80, 110-120
  downloading, 110
  Java, embedding with, 114-120
  scripting
    in Java, 111-112
    interface implementation, 112-114
    JavaAdapter class, 114
RI (Reference Implementation)
  JSR 223 RI, installing, 499-502
robustness, scripting languages, 29-32
run( ) method, 134
running
  BeanShell, 84
  scripts, Groovy, 127-130
runtime performance, code, 26-28

S

safe navigation, GroovyBeans, 168
Safe Navigation listing (4.22), 168
Sample Web Application Descriptor listing (10.1), 454
SAX Handler Example listing (5.25), 230
SaxBuilder, Groovy, 230-231
SaxBuilder listing (5.26), 231
script context
  attributes, 419-423
  binding, Scripting API, 416-428
  namespaces, 416-419
script directory, Web Scripting Framework, 457-458
script files
  BSF (Bean Scripting Framework), executing, 257-259
  Groovy, evaluating, 129-130
Script Job Example listing (7.18), 330
script methods, Web Scripting Framework, 458

script object factory pattern, 354
consequences, 356
problem, 355
related patterns, 359
sample code, 356-359
solution, 355
script( ) method, 95
Scripted Component Pattern Example listing (8.1), 340
scripted components pattern, 337
consequences, 339
problem, 337-338
related patterns, 341
sample code, 340-341
solution, 338-339
Scripted Object Factory Pattern Example listing (8.11), 357
ScriptEngineManager class, shortcut methods, 414-415
ScriptEngineManager's Shortcut Methods listing (9.17), 414
ScriptException, handling, 403-404
scripting. *See also* programming
administration, 328-334
BeanShell, 83-98
autoboxing, 89-90
basic syntax, 86-87
commands, 91
embedding, 94-98
for loop, 88-89
interface implementation, 93-94
JavaBeans, 88
loosely typed syntax, 87
methods, 91-92
objects, 92
switch-case statement, 90
Workspace Editor, 85
design patterns, 335-337
active file pattern, 375-380
extension point pattern, 369-374
mediator pattern, 341-354
observer pattern, 359-369
script object factory pattern, 354-359
scripted components pattern, 337-341
Jython, 98-109
management, 328-334
Python, 98-109
basic syntax, 101-103
embedding, 108-109
exception handling, 107-108
in Java, 103-105
interface implementation, 105-107
Rhino, 110-120
embedding, 114-120
in Java, 111-112
interface implementation, 112-114
JavaAdapter class, 114
shell scripting, 323-328
system programming, 26-35
hybrids, 35-36
Web Scripting Framework, disabling, 456-457
Scripting API, 94, 385, 388-391
architecture, 391
binding, 404-411
dynamic binding, 442-444
engine scope, 405-411
global scope, 411-416
script context, 416-428
BSF, compared, 445
code generation, 428
method call syntax, 429-431
output statement, 429
programs, 431-432
discovery mechanism, 391-393
engine interfaces, 432
Compilable interface, 437-440
Incovable interface, 432-437
engine metadata, 393, 395
General Scripting API, 389
origins of, 386-388
script, evaluating, 400-404

scripting engines
creating, 395-398
registering, 399
threading, 440-442
Web Scripting API, 389
Web Scripting Framework, 447, 453-456
architecture, 448-453, 482-488
bindings, 462-469
configuring, 456-462
forward( ) method, 471-473
include( ) method, 469-471
language tags, 478-481
session handling, 473-478
threading, 481-482

scripting engines
creating, Scripting API, 395-398
registering, Scripting API, 399

scripting environments, BSF (Bean Scripting Framework), 245-246
applications, 275-285, 287
architecture, 248-249
bean declaration, 268-270
bean registration, 265-268
compilation, 270-275
data binding, 264-270
downloading, 247
engine initialization, 252
exception handling, 255
functions, 259-264
manager initialization, 252
methods, 259-264
origins of, 247
script files, 257-259
scripting language registration, 249-252
scripts, 253-257

scripting languages, 4-24. *See also* programming languages
administration, 55-58
BeanShell, 83-98, 126
closures, 20
code as data, 19-23
compilers, 8-12
customization, 49-51
data structures, 17-19
embedding, 70
extensiblity, 70-71
functions, as method arguments, 21-23
Groovy, 120-121, 125, 194-195
advantages of, 126
configuring, 492
downloading, 491
Groovlets, 212-219
GroovySQL, 196-212
IDE support, 495-497
installing, 127, 491-492
Markup syntax, 223-236
Swing user interfaces, 236-243
templates, 220-223
testing, 492-493
interpreters, 8-12
Java
API (application programming interface), 80-82
architecture, 80-82
class files, 80
JVM (Java Virtual Machine), 80-82
programming language, 80
JRuby, 122
JudoScript, 122
JVM (Java Virtual Machine), using in, 82-83
Jython, 98-109, 126
learnability, 71-72
management, 55-58
ObjectScript, 123
Perl, 43
prototyping models, 44-48
evolutionary prototyping, 46
Pyhton, 47-48
throwaway prototyping, 45-46
Pyhton, 126
Python, 18, 98-109
registering, BSF (Bean Scripting Framework), 249-252

Rhino, 110-114, 116-120
software development support, 51-55
project building, 51-53
testing, 53, 55
source code, production environment, 12-13
Tcl (Tool Command Language), 37-38, 43-44
Tcl/Java, 122
typing, 13, 15
composite types, 14
dynamic typing, 15-16
native types, 14
user-defined types, 14
weak typing, 17
UNIX shell languages, 41
user interface programming, 58-59
virtual machines, 24-25
web applications, 59, 61-67
ASP (Activer Server Pages), 64
games, 68-69
JavaScript, 65-67
Perl, 61-62
PHP, 62-64
UNIX, 68
wiring, 40-44
UNIX shell languages, 42
scripts
evaluating, Scripting API, 400-404
Groovy
command-line arguments, 136
compiling, 130-133
dependencies, 131
running, 127-130
structure, 133-136
unit testing, 303-304
working with, BSF (Bean Scripting Framework), 253-257
SCUMM (Script Creation Utility for Maniac Mansion), 69
second-generation programming languages, 5
security, Groovy, 190-194
Security Example listing (4.47), 191
Security Policy listing (4.48), 191
select( ) method, 22
sendError( ) method, 468
servers, web servers, Apache Web server, 62
servlets, Web Scripting Framework architecture, 449-451
session handling, Web Scripting Framework, 473-478
Session Handling—Login Form (form.jsp) listing (10.7), 474
Session Handling—Login Servlet listing (10.8), 475
Session Handling—PHP Script listing (10.9), 477
session scope, Web environments, 448
session variable (Groovlet), 215
set( ) method, 95
setBindings( ) method, 407
setChanged( ) method, 363
setClassPath( ) method, 324
setEmail( ) method, 308
setPrice( ) method, 363
setting classpaths, Groovy, 131-132
setUp( ) method, 294
shell languages, UNIX, 41-42
shell scripting, 323-328
Shell Scripting Example listing (7.16), 327
shortcut methods, ScriptEngineManager class, 414-415
shouldFail( ) method, 299
show( ) method, 350
showDialog( ) method, 346, 350
Simple XSLT Transformation listing (6.21), 281
single inheritance model, Java, 78
sleep( ) method, 444
Smith, Ben, 61

software development support, scripting languages, 51-55
  project building, 51-53
  testing, 53-55
solutions
  active file pattern, 375
  extension point pattern, 370
  mediator scripting pattern, 342-345
  observer pattern, 360-362
  script object factory pattern, 355
  scripted components pattern, 338-339
someFunc( ) function, 103
source code, production environment, 12-13
source( ) method, 95-96
splitEachLine( ) method, Groovy, 180
splitEachLine( ) Method listing (4.35), 180
SQL Object Creation Alternatives listing (5.2), 199
Sql.loadDriver( ) method, GroovySQL, 199-200
Sql.loadDriver( ) Method listing (5.3), 199
standalone methods, 91, 134
statements
  Groovy, 138
  prepared statements, GroovySQL, 206-207
  switch-case statement, BeanShell, 90
Stored Procedure Call listing (5.12), 208
Stored Procedure listing (5.11), 207
stored procedures, GroovySQL, 207-208
strings, Groovy, 143-145
  GStrings, 145-146
strongly typed languages, 17
structure, scripts, Groovy, 133, 135-136
Subject Component listing (8.12), 363
Swing user interfaces
  TableLayout component, 239-240
  TableModel component, 241-243
SwingBuilder, Groovy, 236-243
SwingBuilder listing (5.31), 237
switch-case statement, BeanShell, 90
switch-case structure, Groovy, 154-156
switch-case Structure listing (4.14), 154
switches, jythonc, 101
syntax
  BeanShell
    autoboxing, 89-90
    basic syntax, 86-87
    for loop, 88-89
    JavaBeans, 88
    loosely typed syntax, 87
    switch-case statement, 90
  Groovy, 137-177
  Python, basic syntax, 101-103
system operations, Groovy, 178
  files, 178-182
  processes, 182-183
system programming, scripting
  compared, 26-35
  hybrids, 35-36

T

TableLayout component (Swing), 239-240
TableLayout listing (5.32), 239
TableModel component (Swing), 241-243
TableModel listing (5.33), 241
Tag Library Descriptor (TLD) file, 278
tags, language tags, Web Scripting Framework, 478-481
Tax Observer Label Widget listing (8.15), 365
Tcl (Tool Command Language), 37-38, 43-44
  Tk extension, 58-59
Tcl Blend, 122
Tcl/Java, 122
TDD (test-driven development), 33
Template Example listing (5.20), 222
templates, Groovy templates, 220-223
termination( ) method, 134
test suites, unit testing, 300-303
test-driven development (TDD), 33
testAnimal( ) method, 156

testing
  Groovy, 492-493
  scripting languages, 53-55
testInit( ) method, 294
testIteration( ) method, 294
TestSuite Example listing (7.9), 300
Thinking in Java, 30
third-generation programming languages, 6
Thomas, Dave, 53
threading
  Java, 77
  Scripting API, 440-442
  Web Scripting Framework, 481-482
Threading listing (9.36), 441
throwaway prototyping, 45-46
Tk extension (Tcl), 58-59
TLD (Tag Library Descriptor) file, 278
Tool Command Language (Tcl). *See* Tcl (Tool Command Language)
transactions, GroovySQL, 208-209
Transactions listing (5.13), 208
triple-quote syntax, Groovy, 144-145
Triple-Quote Syntax listing (4.8), 144
type juggling, Groovy, 140-143
type theory, 13
types, classes, 14
typing
  dynamic typing, Groovy, 139-140
  loose typing, Groovy, 138-140
  programming languages, 13, 15
    composite types, 14
    dynamic typing, 15-16
    native types, 14
    user-defined types, 14
    weak typing, 17, 138

U

unified resource locators (URLs), 60
unit testing, 292-293
  assertion methods, 297-300
  GroovyTestCase class, 296-297
  JUnit, 293-295
  scripts, 303-304
  test suites, 300-303
UNIX
  programming, 40
  scripting, 68
  shell languages, 41-42
unregisterBean( ) method, 266
update( ) method, 238, 364
updateObservers( ) method, 364
upper( ) function, 102
URLs (unified resource locators), 60
user interface programming, 58-59
user manuals, BeanShell, 98
user-defined class loaders (JVM), 82
user-defined types, programming languages, 14
Using Custom Application Context listing (9.27), 426

V

valueChanged( ) method, 348
van Rossum, Guido, 32, 48
variables
  global scope, overriding, 413-414
  Groovlets, 214
Variables Overriding—An Example listing (9.16), 413
VBA (Visual Basic for Applications), 50-51
Venners, Bill, 36
virtual machines, 7
  scripting languages, 24-25
Visual Basic for Applications (VBA), 50-51

W

waitFor( ) method, 183
weak typing, 17, 138
weakly typed languages, 17
web applications, 59, 61-67
  ASP (Activer Server Pages), 64
  games, 68-69
  JavaScript, 65-67
  Perl, 61-62

PHP, 62-64
UNIX, 68
Web Scripting API, 389
Web Scripting Framework, 447, 453-456
architecture, 448, 482-488
context, 448-449
interaction, 451-453
servlet, 449-451
bindings, 462
application, 462-464
request handling, 464-467
response, 468
servlet, 468-469
configuring, 456-462
forward( ) method, 471-473
include( ) method, 469-471
language tags, 478-481
languages, allowing, 459
results, displaying, 460-462
script directory, 457-458
script methods, 458
scripting, disabling, 456-457
session handling, 473-478
threading, 481-482
web servers, Apache Web servers, 62
widgetChanged( ) method, 346, 348, 350
wiring scripting languages, 40-44
Perl, 43
Tcl (Tool Command Language), 43-44
UNIX shell languages, 41-42
Workspace Editor, BeanShell, 85
write( ) method, Groovy, 181
write( ) Method listing (4.38), 181
writers, engine writers, 427-428
WWW (World Wide Web), 59
HTML (Hypertext Markup Language), 60
HTTP (HyperText Transfer Protocol), 60
hypertext, 60
URLs (unified resource locators), 60

X–Z

Xalan-J (XSLT), BSF (Bean Scripting Framework), 280-287
Xalan-J BSF Example listing (6.24), 285
Xalan-J Extension Example listing (6.23), 283
XHTML Markup Example listing (5.23), 226
XPath (XML Path Language), 280
XSLT (Extensible Stylesheet Language Transformation), 280

Register Your Book

at www.awprofessional.com/register

You may be eligible to receive:

- Advance notice of forthcoming editions of the book
- Related book recommendations
- Chapter excerpts and supplements of forthcoming titles
- Information about special contests and promotions throughout the year
- Notices and reminders about author appearances, tradeshows, and online chats with special guests

Contact us

If you are interested in writing a book or reviewing manuscripts prior to publication, please write to us at:

Editorial Department
Addison-Wesley Professional
75 Arlington Street, Suite 300
Boston, MA 02116 USA
Email: AWPro@aw.com

Addison-Wesley

Visit us on the Web: http://www.awprofessional.com