REVISED AND UPDATED FOR JAVA SE 7

PRENTICE HALL

CORE JAVA®

Volume II—Advanced Features

NINTH EDITION

CAY S. HORSTMANN · GARY CORNELL

Core Java®

Volume II—Advanced Features

Ninth Edition

This page intentionally left blank

Core Java®

Volume II—Advanced Features

Ninth Edition

Cay S. Horstmann
Gary Cornell

PRENTICE HALL

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

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@pearson.com

Visit us on the Web: informit.com/ph

Library of Congress Cataloging-in-Publication Data:

Horstmann, Cay S., 1959-
Core Java / Cay S. Horstmann, Gary Cornell.—Ninth edition.
pages cm
Includes index.
ISBN 978-0-13-708189-9 (v. 1 : pbk. : alk. paper) 1. Java (Computer program language) I. Cornell, Gary. II. Title.
QA76.73.J38H6753 2013
005.13'3—dc23

2012035397

ISBN-13: 978-0-13-708160-8
ISBN-10: 0-13-708160-X
Text printed in the United States on recycled paper at Edwards Brothers Malloy in Ann Arbor, Michigan.
Second printing, September 2013

Contents

Preface

To the Reader

The book you have in your hands is the second volume of the ninth edition of *Core Java*®, fully updated for Java SE 7. The first volume covers the essential features of the language; this volume deals with the advanced topics that a programmer needs to know for professional software development. Thus, as with the first volume and the previous editions of this book, we are still targeting programmers who want to put Java technology to work on real projects.

Please note: If you are an experienced developer who is comfortable with advanced language features such as inner classes and generics, you need not have read the first volume in order to benefit from this volume. While we do refer to sections of the previous volume when appropriate (and, of course, hope you will buy or have bought Volume I), you can find all the background material you need in any comprehensive introductory book about the Java platform.

Finally, as is the case with any book, errors and inaccuracies are inevitable. Should you find any in this book, we would very much like to hear about them. Of course, we would prefer to hear about them only once. For this reason, we have put up a web site at http://horstmann.com/corejava with a FAQ, bug fixes, and workarounds. Strategically placed at the end of the bug report web page (to encourage you to read the previous reports) is a form that you can use to report bugs or problems and to send suggestions for improvements to future editions.

About This Book

The chapters in this book are, for the most part, independent of each other. You should be able to delve into whatever topic interests you the most and read the chapters in any order.

The topic of **Chapter 1** is input and output handling (I/O). In Java, all I/O is handled through so-called streams. Streams let you deal, in a uniform manner, with communications among various sources of data, such as files, network connections, or memory blocks. We include detailed coverage of the reader and writer classes that make it easy to deal with Unicode. We show you what goes on under the hood when you use the object serialization mechanism, which makes saving and loading objects easy and convenient. We then move on to regular expressions and the NIO2 library of Java SE 7, which makes common operations (such as reading all lines in a file) very convenient.

Chapter 2 covers XML. We show you how to parse XML files, how to generate XML, and how to use XSL transformations. As a useful example, we show you how to specify the layout of a Swing form in XML. We also discuss the XPath API, which makes "finding needles in XML haystacks" much easier.

Chapter 3 covers the networking API. Java makes it phenomenally easy to do complex network programming. We show you how to make network connections to servers, how to implement your own servers, and how to make HTTP connections.

Chapter 4 covers database programming. The main focus is on JDBC, the Java database connectivity API that lets Java programs connect to relational databases. We show you how to write useful programs to handle realistic database chores, using a core subset of the JDBC API. (A complete treatment of the JDBC API would require a book almost as long as this one.) We finish the chapter with a brief introduction into hierarchical databases and discuss JNDI (the Java Naming and Directory Interface) and LDAP (the Lightweight Directory Access Protocol).

Chapter 5 discusses a feature that we believe can only grow in importance: internationalization. The Java programming language is one of the few languages designed from the start to handle Unicode, but the internationalization support in the Java platform goes much further. As a result, you can internationalize Java applications so that they not only cross platforms but cross country boundaries as well. For example, we show you how to write a retirement calculator that uses either English, German, or Chinese languages.

Chapter 6 contains all the Swing material that didn't make it into Volume I, especially the important but complex tree and table components. We show the basic uses of editor panes, the Java implementation of a "multiple document" interface, progress indicators used in multithreaded programs, and "desktop integration features" such as splash screens and support for the system tray. Again, we focus on the most useful constructs that you are likely to encounter in practical programming because an encyclopedic coverage of the entire Swing library would fill several volumes and would only be of interest to dedicated taxonomists.

Chapter 7 covers the Java 2D API, which you can use to create realistic drawings and special effects. The chapter also covers some advanced features of the AWT (Abstract Windowing Toolkit) that seemed too specialized for coverage in Volume I but should, nonetheless, be part of every programmer's toolkit. These features include printing and the APIs for cut-and-paste and drag-and-drop.

Chapter 8 explains what you need to know about the component API for the Java platform—JavaBeans. We show you how to write your own beans that other programmers can manipulate in integrated builder environments. We conclude this chapter by showing you how you can use JavaBeans persistence to store your data in a format that—unlike object serialization—is suitable for long-term storage.

Chapter 9 takes up the Java security model. The Java platform was designed from the ground up to be secure, and this chapter takes you under the hood to see how this design is implemented. We show you how to write your own class loaders and security managers for special-purpose applications. Then, we take up the security API that allows for such important features as message and code signing, authorization and authentication, and encryption. We conclude with examples that use the AES and RSA encryption algorithms.

Chapter 10 covers distributed objects. We cover RMI (Remote Method Invocation) in detail. This API lets you work with Java objects that are distributed over multiple machines.

Chapter 11 discusses three techniques for processing code. The scripting and compiler APIs allow your program to call code in scripting languages such as JavaScript or Groovy, and to compile Java code. Annotations allow you to add arbitrary information (sometimes called metadata) to a Java program. We show you how annotation processors can harvest these annotations at the source or class file level, and how annotations can be used to influence the behavior of classes at runtime. Annotations are only useful with tools, and we hope that our discussion will help you select useful annotation processing tools for your needs.

Chapter 12 takes up native methods, which let you call methods written for a specific machine such as the Microsoft Windows API. Obviously, this feature is controversial: Use native methods, and the cross-platform nature of the Java platform vanishes. Nonetheless, every serious programmer writing Java applications for specific platforms needs to know these techniques. At times, you need to turn to the operating system's API for your target platform when you interact with a device or service that is not supported by Java. We illustrate this by showing you how to access the registry API in Windows from a Java program.

As always, all chapters have been completely revised for the latest version of Java. Outdated material has been removed, and the new APIs of Java SE 7 are covered in detail.

Conventions

As is common in many computer books, we use `monospace type` to represent computer code.

NOTE: Notes are tagged with "note" icons that look like this.

TIP: Tips are tagged with "tip" icons that look like this.

CAUTION: When there is danger ahead, we warn you with a "caution" icon.

C++ NOTE: There are a number of C++ notes that explain the difference between the Java programming language and C++. You can skip them if you aren't interested in C++.

Java comes with a large programming library, or Application Programming Interface (API). When using an API call for the first time, we add a short summary description at the end of the section. These descriptions are a bit more informal but, we hope, also a little more informative than those in the official online API documentation. The names of interfaces are in italics, just like in the official documentation. The number after a class, interface, or method name is the JDK version in which the feature was introduced.

`Application Programming Interface` 1.2

Programs whose source code is included in the companion code for this book are listed as examples; for instance,

Listing 1.1 `ScriptTest.java`

You can download the companion code from http://horstmann.com/corejava.

Acknowledgments

Writing a book is always a monumental effort, and rewriting doesn't seem to be much easier, especially with such a rapid rate of change in Java technology. Making a book a reality takes many dedicated people, and it is my great pleasure to acknowledge the contributions of the entire Core Java team.

A large number of individuals at Prentice Hall provided valuable assistance, but they managed to stay behind the scenes. I'd like them all to know how much I appreciate their efforts. As always, my warm thanks go to my editor, Greg Doench, for steering the book through the writing and production process, and for allowing me to be blissfully unaware of the existence of all those folks behind the scenes. I am very grateful to Julie Nahil for production support, and to Dmitry Kirsanov and Alina Kirsanova for copyediting and typesetting the manuscript.

Thanks to the many readers of earlier editions who reported embarrassing errors and made lots of thoughtful suggestions for improvement. I am particularly grateful to the excellent reviewing team that went over the manuscript with an amazing eye for detail and saved me from many more embarrassing errors.

Reviewers of this and earlier editions include Chuck Allison (Contributing Editor, C/C++ Users Journal), Lance Anderson (Oracle), Alec Beaton (PointBase, Inc.), Cliff Berg (iSavvix Corporation), Joshua Bloch, David Brown, Corky Cartwright, Frank Cohen (PushToTest), Chris Crane (devXsolution), Dr. Nicholas J. De Lillo (Manhattan College), Rakesh Dhoopar (Oracle), Robert Evans (Senior Staff, The Johns Hopkins University Applied Physics Lab), David Geary (Sabreware), Jim Gish (Oracle), Brian Goetz (Principal Consultant, Quiotix Corp.), Angela Gordon, Dan Gordon, Rob Gordon, John Gray (University of Hartford), Cameron Gregory (olabs.com), Marty Hall (The Johns Hopkins University Applied Physics Lab), Vincent Hardy, Dan Harkey (San Jose State University), William Higgins (IBM), Vladimir Ivanovic (PointBase), Jerry Jackson (ChannelPoint Software), Tim Kimmet (Preview Systems), Chris Laffra, Charlie Lai, Angelika Langer, Doug Langston, Hang Lau (McGill University), Mark Lawrence, Doug Lea (SUNY Oswego), Gregory Longshore, Bob Lynch (Lynch Associates), Philip Milne (consultant), Mark Morrissey (The Oregon Graduate Institute), Mahesh Neelakanta (Florida Atlantic University), Hao Pham, Paul Philion, Blake Ragsdell, Ylber Ramadani (Ryerson University), Stuart Reges (University of Arizona), Rich Rosen (Interactive Data Corporation), Peter Sanders

(ESSI University, Nice, France), Dr. Paul Sanghera (San Jose State University and Brooks College), Paul Sevinc (Teamup AG), Devang Shah, Richard Slywczak (NASA/Glenn Research Center), Bradley A. Smith, Steven Stelting, Christopher Taylor, Luke Taylor (Valtech), George Thiruvathukal, Kim Topley (author of *Core JFC, Second Edition*), Janet Traub, Paul Tyma (consultant), Peter van der Linden, Burt Walsh, Joe Wang (Oracle), and Dan Xu (Oracle).

Cay Horstmann
San Francisco, California
December 2012

CHAPTER 10

Scripting, Compiling, and Annotation Processing

In this chapter:

This chapter introduces three techniques for processing code. The scripting API lets you invoke code in a scripting language such as JavaScript or Groovy. You can use the compiler API when you want to compile Java code inside your application. Annotation processors operate on Java source or class files that contain annotations. As you will see, there are many applications for annotation processing, ranging from simple diagnostics to "bytecode engineering"—the insertion of bytecodes into class files or even running programs.

10.1 Scripting for the Java Platform

A scripting language is a language that avoids the usual edit/compile/link/run cycle by interpreting the program text at runtime. Scripting languages have a number of advantages:

- Rapid turnaround, encouraging experimentation
- Changing the behavior of a running program
- Enabling customization by program users

On the other hand, most scripting languages lack features that are beneficial for programming complex applications, such as strong typing, encapsulation, and modularity.

It is therefore tempting to combine the advantages of scripting and traditional languages. The scripting API lets you do just that for the Java platform. It enables you to invoke scripts written in JavaScript, Groovy, Ruby, and even exotic languages such as Scheme and Haskell, from a Java program. (The other direction—accessing Java from the scripting language—is the responsibility of the scripting language provider. Most scripting languages that run on the Java virtual machine have this capability.)

In the following sections, we'll show you how to select an engine for a particular language, how to execute scripts, and how to take advantage of advanced features that some scripting engines offer.

10.1.1 Getting a Scripting Engine

A scripting engine is a library that can execute scripts in a particular language. When the virtual machine starts, it discovers the available scripting engines. To enumerate them, construct a `ScriptEngineManager` and invoke the `getEngineFactories` method. You can ask each engine factory for the supported engine names, MIME types, and file extensions. Table 10.1 shows typical values.

Usually, you know which engine you need, and you can simply request it by name, MIME type, or extension. For example:

```
ScriptEngine engine = manager.getEngineByName("JavaScript");
```

Java SE 7 includes a version of Rhino, a JavaScript interpreter developed by the Mozilla foundation. You can add more languages by providing the necessary JAR files on the class path. You will generally need two sets of JAR files. The scripting language itself is implemented by a single JAR file or a set of JARs. The engine that adapts the language to the scripting API usually requires an additional JAR. The site http://java.net/projects/scripting provides engines for a

Table 10.1 Properties of Scripting Engine Factories

Engine	Names	MIME types	Extensions
Rhino (included with Java SE)	js, rhino, JavaScript, javascript, ECMAScript, ecmascript	`application/javascript`, `application/ecmascript`, `text/javascript`, `text/ecmascript`	`js`
Groovy	groovy	None	`groovy`
SISC Scheme	scheme, sisc	None	`scc`, `sce`, `scm`, `shp`

wide range of scripting languages. For example, to add support for Groovy, the class path should contain *groovy*`/lib/*` (from http://groovy.codehaus.org) and `groovy-engine.jar` (from http://java.net/projects/scripting).

`javax.script.ScriptEngineManager` 6

- `List<ScriptEngineFactory> getEngineFactories()`
  gets a list of all discovered engine factories.
- `ScriptEngine getEngineByName(String name)`
- `ScriptEngine getEngineByExtension(String extension)`
- `ScriptEngine getEngineByMimeType(String mimeType)`
  gets the script engine with the given name, script file extension, or MIME type.

***`javax.script.ScriptEngineFactory`* 6**

- `List<String> getNames()`
- `List<String> getExtensions()`
- `List<String> getMimeTypes()`
  gets the names, script file extensions, and MIME types under which this factory is known.

10.1.2 Script Evaluation and Bindings

Once you have an engine, you can call a script simply by invoking

```
Object result = engine.eval(scriptString);
```

If the script is stored in a file, open a `Reader` and call

```
Object result = engine.eval(reader);
```

You can invoke multiple scripts on the same engine. If one script defines variables, functions, or classes, most scripting engines retain the definitions for later use. For example,

```
engine.eval("n = 1728");
Object result = engine.eval("n + 1");
```

will return `1729`.

NOTE: To find out whether it is safe to concurrently execute scripts in multiple threads, call

```
Object param = factory.getParameter("THREADING");
```

The returned value is one of the following:

- `null`: Concurrent execution is not safe.
- `"MULTITHREADED"`: Concurrent execution is safe. Effects from one thread might be visible from another thread.
- `"THREAD-ISOLATED"`: In addition to `"MULTITHREADED"`, different variable bindings are maintained for each thread.
- `"STATELESS"`: In addition to `"THREAD-ISOLATED"`, scripts do not alter variable bindings.

You will often want to add variable bindings to the engine. A binding consists of a name and an associated Java object. For example, consider these statements:

```
engine.put(k, 1728);
Object result = engine.eval("k + 1");
```

The script code reads the definition of `k` from the bindings in the "engine scope." This is particularly important because most scripting languages can access Java objects, often with a syntax that is simpler than the Java syntax. For example,

```
engine.put(b, new JButton());
engine.eval("b.text = 'Ok'");
```

Conversely, you can retrieve variables that were bound by scripting statements:

```
engine.eval("n = 1728");
Object result = engine.get("n");
```

In addition to the engine scope, there is also a global scope. Any bindings that you add to the `ScriptEngineManager` are visible to all engines.

Instead of adding bindings to the engine or global scope, you can collect them in an object of type `Bindings` and pass it to the `eval` method:

```
Bindings scope = engine.createBindings();
scope.put(b, new JButton());
engine.eval(scriptString, scope);
```

This is useful if a set of bindings should not persist for future calls to the `eval` method.

NOTE: You might want to have scopes other than the engine and global scopes. For example, a web container might need request and session scopes. However, then you are on your own. You will need to write a class that implements the `ScriptContext` interface, managing a collection of scopes. Each scope is identified by an integer number, and scopes with lower numbers should be searched first. (The standard library provides a `SimpleScriptContext` class, but it only holds global and engine scopes.)

javax.script.ScriptEngine 6

- `Object eval(String script)`
- `Object eval(Reader reader)`
- `Object eval(String script, Bindings bindings)`
- `Object eval(Reader reader, Bindings bindings)`

  evaluates the script given by the string or reader, subject to the given bindings.
- `Object get(String key)`
- `void put(String key, Object value)`

  gets or puts a binding in the engine scope.
- `Bindings createBindings()`

  creates an empty `Bindings` object suitable for this engine.

javax.script.ScriptEngineManager 6

- `Object get(String key)`
- `void put(String key, Object value)`

  gets or puts a binding in the global scope.

javax.script.Bindings 6

- `Object get(String key)`
- `void put(String key, Object value)`

  gets or puts a binding into the scope represented by this `Bindings` object.

10.1.3 Redirecting Input and Output

You can redirect the standard input and output of a script by calling the `setReader` and `setWriter` methods of the script context. For example,

```
StringWriter writer = new StringWriter();
engine.getContext().setWriter(new PrintWriter(writer, true));
```

Any output written with the JavaScript `print` or `println` functions is sent to `writer`.

CAUTION: You can pass any `Writer` to the `setWriter` method, but the Rhino engine throws an exception if it is not a `PrintWriter`.

The `setReader` and `setWriter` methods only affect the scripting engine's standard input and output sources. For example, if you execute the JavaScript code

```
println("Hello");
java.lang.System.out.println("World");
```

only the first output is redirected.

The Rhino engine does not have the notion of a standard input source. Calling `setReader` has no effect.

javax.script.ScriptEngine 6

- `ScriptContext getContext()`
  gets the default script context for this engine.

javax.script.ScriptContext 6

- `Reader getReader()`
- `void setReader(Reader reader)`
- `Writer getWriter()`
- `void setWriter(Writer writer)`
- `Writer getErrorWriter()`
- `void setErrorWriter(Writer writer)`
  gets or sets the reader for input or writer for normal or error output.

10.1.4 Calling Scripting Functions and Methods

With many script engines, you can invoke a function in the scripting language without having to evaluate the actual script code. This is useful if you allow users to implement a service in a scripting language of their choice.

The script engines that offer this functionality implement the `Invocable` interface. In particular, the Rhino engine implements `Invocable`.

To call a function, call the `invokeFunction` method with the function name, followed by the function parameters:

```
if (engine implements Invocable)
   ((Invocable) engine).invokeFunction("aFunction", param1, param2);
```

If the scripting language is object-oriented, you call can a method like this:

```
((Invocable) engine).invokeMethod(implicitParam, "aMethod", explicitParam1, explicitParam2);
```

Here, the `implicitParam` object is a proxy to an object in the scripting language. It must be the result of a prior call to the scripting engine.

NOTE: If the script engine does not implement the `Invocable` interface, you might still be able to call a method in a language-independent way. The `getMethodCallSyntax` method of the `ScriptEngineFactory` interface produces a string that you can pass to the `eval` method. However, all method parameters must be bound to names, whereas `invokeMethod` can be called with arbitrary values.

You can go a step further and ask the scripting engine to implement a Java interface. Then you can call scripting functions and methods with the Java method call syntax.

The details depend on the scripting engine, but typically you need to supply a function for each method of the interface. For example, consider a Java interface

```
public interface Greeter
{
   String greet(String whom);
}
```

In Rhino, you provide a function

```
function greet(x) { return "Hello, " + x + "!"; }
```

This code must be evaluated first. Then you can call

```
Greeter g = ((Invocable) engine).getInterface(Greeter.class);
```

Now you can make a plain Java method call

```
String result = g.greet("World");
```

Behind the scenes, the JavaScript `greet` method is invoked. This approach is similar to making a remote method call, as discussed in Chapter 11.

In an object-oriented scripting language, you can access a script class through a matching Java interface. For example, consider this JavaScript code, which defines a `SimpleGreeter` class.

```
function SimpleGreeter(salutation) { this.salutation = salutation; }
SimpleGreeter.prototype.greet = function(whom) { return this.salutation + ", " + whom + "!"; }
```

You can use this class to construct greeters with different salutations (such as "Hello", "Goodbye", and so on).

NOTE: For more information on how to define classes in JavaScript, see *JavaScript—The Definitive Guide, Fifth Edition*, by David Flanagan (O'Reilly, 2006).

After evaluating the JavaScript class definition, call

```
Object goodbyeGreeter = engine.eval("new SimpleGreeter('Goodbye')");
Greeter g = ((Invocable) engine).getInterface(goodbyeGreeter, Greeter.class);
```

When you call `g.greet("World")`, the `greet` method is invoked on the JavaScript object `goodbyeGreeter`. The result is a string `"Goodbye, World!"`.

In summary, the `Invocable` interface is useful if you want to call scripting code from Java without worrying about the scripting language syntax.

javax.script.Invocable 6

- `Object invokeFunction(String name, Object... parameters)`
- `Object invokeMethod(Object implicitParameter, String name, Object... explicitParameters)`
  invokes the function or method with the given name, passing the given parameters.
- `<T> T getInterface(Class<T> iface)`
  returns an implementation of the given interface, implementing the methods with functions in the scripting engine.
- `<T> T getInterface(Object implicitParameter, Class<T> iface)`
  returns an implementation of the given interface, implementing the methods with the methods of the given object.

10.1.5 Compiling a Script

Some scripting engines can compile scripting code into an intermediate form for efficient execution. Those engines implement the `Compilable` interface. The following example shows how to compile and evaluate code contained in a script file:

```
Reader reader = new FileReader("myscript.js");
CompiledScript script = null;
if (engine implements Compilable)
   CompiledScript script = ((Compilable) engine).compile(reader);
```

Once the script is compiled, you can execute it. The following code executes the compiled script if compilation was successful, or the original script if the engine didn't support compilation.

```
if (script != null)
   script.eval();
else
   engine.eval(reader);
```

Of course, it only makes sense to compile a script if you need to execute it repeatedly.

`javax.script.Compilable` 6

- `CompiledScript compile(String script)`
- `CompiledScript compile(Reader reader)`

  compiles the script given by a string or reader.

`javax.script.CompiledScript` 6

- `Object eval()`
- `Object eval(Bindings bindings)`

  evaluates this script.

10.1.6 An Example: Scripting GUI Events

To illustrate the scripting API, we will write a sample program that allows users to specify event handlers in a scripting language of their choice.

Have a look at the program in Listing 10.1 that adds scripting to an arbitrary frame class. By default it reads the `ButtonFrame` class in Listing 10.2, which is similar to the event handling demo in Volume I, with two differences:

- Each component has its `name` property set.
- There are no event handlers.

The event handlers are defined in a property file. Each property definition has the form

```
componentName.eventName = scriptCode
```

For example, if you choose to use JavaScript, supply the event handlers in a file `js.properties`, like this:

```
yellowButton.action=panel.background = java.awt.Color.YELLOW
blueButton.action=panel.background = java.awt.Color.BLUE
redButton.action=panel.background = java.awt.Color.RED
```

The companion code also has files for Groovy and SISC Scheme.

The program starts by loading an engine for the language specified on the command line. If no language is specified, we use JavaScript.

We then process a script `init.`*language* if it is present. This seems like a good idea in general; moreover, the Scheme interpreter needs some cumbersome initializations that we did not want to include in every event handler script.

Next, we recursively traverse all child components and add the bindings (*name*, *object*) into the engine scope.

Then we read the file *language*`.properties`. For each property, we synthesize an event handler proxy that causes the script code to be executed. The details are a bit technical. You might want to read the section on proxies in Volume I, Chapter 6, together with the section on JavaBeans events in Chapter 8 of this volume, if you want to follow the implementation in detail. The essential part, however, is that each event handler calls

```
engine.eval(scriptCode);
```

Let us look at the `yellowButton` in more detail. When the line

```
yellowButton.action=panel.background = java.awt.Color.YELLOW
```

is processed, we find the `JButton` component with the name `"yellowButton"`. We then attach an `ActionListener` with an `actionPerformed` method that executes the script

```
panel.background = java.awt.Color.YELLOW
```

The engine contains a binding that binds the name `"panel"` to the `JPanel` object. When the event occurs, the `setBackground` method of the panel is executed, and the color changes.

You can run this program with the JavaScript event handlers, simply by executing

```
java ScriptTest
```

For the Groovy handlers, use

```
java -classpath .:groovy/lib/*:jsr223-engines/groovy/build/groovy-engine.jar ScriptTest groovy
```

Here, *groovy* is the directory into which you installed Groovy, and *jsr223-engines* is the directory that contains the engine adapters from http://java.net/projects/scripting.

To try out Scheme, download SISC Scheme from http://sisc-scheme.org and run

```
java -classpath .:sisc/*:jsr223-engines/scheme/build/scheme-engine.jar ScriptTest scheme
```

This application demonstrates how to use scripting for Java GUI programming. One could go a step further and describe the GUI with an XML file, as you have seen in Chapter 2. Then our program would become an interpreter for GUIs that have visual presentation defined by XML and behavior defined by a scripting language. Note the similarity to a dynamic HTML page or a dynamic server-side scripting environment.

Listing 10.1 `script/ScriptTest.java`

```java
package script;

import java.awt.*;
import java.beans.*;
import java.io.*;
import java.lang.reflect.*;
import java.util.*;
import javax.script.*;
import javax.swing.*;

/**
 * @version 1.01 2012-01-28
 * @author Cay Horstmann
 */
public class ScriptTest
{
   public static void main(final String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               try
               {
                  ScriptEngineManager manager = new ScriptEngineManager();
```

(Continues)

Listing 10.1 *(Continued)*

```
                  String language;
                  if (args.length == 0)
                  {
                     System.out.println("Available factories: ");
                     for (ScriptEngineFactory factory : manager.getEngineFactories())
                        System.out.println(factory.getEngineName());

                     language = "js";
                  }
                  else language = args[0];

                  final ScriptEngine engine = manager.getEngineByName(language);
                  if (engine == null)
                  {
                     System.err.println("No engine for " + language);
                     System.exit(1);
                  }

                  final String frameClassName = args.length < 2 ? "buttons1.ButtonFrame" : args[1];

                  JFrame frame = (JFrame) Class.forName(frameClassName).newInstance();
                  InputStream in = frame.getClass().getResourceAsStream("init." + language);
                  if (in != null) engine.eval(new InputStreamReader(in));
                  getComponentBindings(frame, engine);

                  final Properties events = new Properties();
                  in = frame.getClass().getResourceAsStream(language + ".properties");
                  events.load(in);

                  for (final Object e : events.keySet())
                  {
                     String[] s = ((String) e).split("\\.");
                     addListener(s[0], s[1], (String) events.get(e), engine);
                  }
                  frame.setTitle("ScriptTest");
                  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                  frame.setVisible(true);
               }
               catch (ReflectiveOperationException | IOException
                  | ScriptException | IntrospectionException ex)
               {
                  ex.printStackTrace();
               }
            }
         });
   }
```

```
    /**
     * Gathers all named components in a container.
     * @param c the component
     * @param namedComponents
     */
    private static void getComponentBindings(Component c, ScriptEngine engine)
    {
       String name = c.getName();
       if (name != null) engine.put(name, c);
       if (c instanceof Container)
       {
          for (Component child : ((Container) c).getComponents())
             getComponentBindings(child, engine);
       }
    }

    /**
     * Adds a listener to an object whose listener method executes a script.
     * @param beanName the name of the bean to which the listener should be added
     * @param eventName the name of the listener type, such as "action" or "change"
     * @param scriptCode the script code to be executed
     * @param engine the engine that executes the code
     * @param bindings the bindings for the execution
     * @throws IntrospectionException
     */
    private static void addListener(String beanName, String eventName, final String scriptCode,
       final ScriptEngine engine) throws ReflectiveOperationException, IntrospectionException
    {
       Object bean = engine.get(beanName);
       EventSetDescriptor descriptor = getEventSetDescriptor(bean, eventName);
       if (descriptor == null) return;
       descriptor.getAddListenerMethod().invoke(bean,
          Proxy.newProxyInstance(null, new Class[] { descriptor.getListenerType() },
             new InvocationHandler()
                {
                   public Object invoke(Object proxy, Method method, Object[] args)
                         throws Throwable
                   {
                      engine.eval(scriptCode);
                      return null;
                   }
                }));
    }

    private static EventSetDescriptor getEventSetDescriptor(Object bean, String eventName)
       throws IntrospectionException
    {
```

(Continues)

Listing 10.1 *(Continued)*

```
      for (EventSetDescriptor descriptor : Introspector.getBeanInfo(bean.getClass())
            .getEventSetDescriptors())
         if (descriptor.getName().equals(eventName)) return descriptor;
      return null;
   }
}
```

Listing 10.2 buttons1/ButtonFrame.java

```
package buttons1;

import javax.swing.*;

public class ButtonFrame extends JFrame
{
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;

   private JPanel panel;
   private JButton yellowButton;
   private JButton blueButton;
   private JButton redButton;

   public ButtonFrame()
   {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      panel = new JPanel();
      panel.setName("panel");
      add(panel);

      yellowButton = new JButton("Yellow");
      yellowButton.setName("yellowButton");
      blueButton = new JButton("Blue");
      blueButton.setName("blueButton");
      redButton = new JButton("Red");
      redButton.setName("redButton");

      panel.add(yellowButton);
      panel.add(blueButton);
      panel.add(redButton);
   }
}
```

10.2 The Compiler API

In the preceding sections, you saw how to interact with code in a scripting language. Now we turn to a different scenario: Java programs that compile Java code. There are quite a few tools that need to invoke the Java compiler, such as:

- Development environments
- Java teaching and tutoring programs
- Build and test automation tools
- Templating tools that process snippets of Java code, such as JavaServer Pages (JSP)

In the past, applications invoked the Java compiler by calling undocumented classes in the *jdk*`/lib/tools.jar` library. As of Java SE 6, a public API for compilation is a part of the Java platform, and it is no longer necessary to use `tools.jar`. This section explains the compiler API.

10.2.1 Compiling the Easy Way

It is very easy to invoke the compiler. Here is a sample call:

```
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
OutputStream outStream = ...;
OutputStream errStream = ...;
int result = compiler.run(null, outStream, errStream, "-sourcepath", "src", "Test.java");
```

A result value of `0` indicates successful compilation.

The compiler sends output and error messages to the provided streams. You can set these parameters to `null`, in which case `System.out` and `System.err` are used. The first parameter of the `run` method is an input stream. As the compiler takes no console input, you can always leave it as `null`. (The `run` method is inherited from a generic `Tool` interface, which allows for tools that read input.)

The remaining parameters of the `run` method are simply the arguments that you would pass to `javac` if you invoked it on the command line. These can be options or file names.

10.2.2 Using Compilation Tasks

You can have even more control over the compilation process with a `CompilationTask` object. In particular, you can

- Control the source of program code—for example, by providing code in a string builder instead of a file.

- Control the placement of class files—for example, by storing them in a database.
- Listen to error and warning messages as they occur during compilation.
- Run the compiler in the background.

The location of source and class files is controlled by a `JavaFileManager`. It is responsible for determining `JavaFileObject` instances for source and class files. A `JavaFileObject` can correspond to a disk file, or it can provide another mechanism for reading and writing its contents.

To listen to error messages, install a `DiagnosticListener`. The listener receives a `Diagnostic` object whenever the compiler reports a warning or error message. The `DiagnosticCollector` class implements this interface. It simply collects all diagnostics so that you can iterate through them after the compilation is complete.

A `Diagnostic` object contains information about the problem location (including file name, line number, and column number) as well as a human-readable description.

To obtain a `CompilationTask` object, call the `getTask` method of the `JavaCompiler` class. You need to specify:

- A `Writer` for any compiler output that is not reported as a `Diagnostic`, or `null` to use `System.err`
- A `JavaFileManager`, or `null` to use the compiler's standard file manager
- A `DiagnosticListener`
- Option strings, or `null` for no options
- Class names for annotation processing, or `null` if none are specified (we'll discuss annotation processing later in this chapter)
- `JavaFileObject` instances for source files

You need to provide the last three arguments as `Iterable` objects. For example, a sequence of options might be specified as

```
Iterable<String> options = Arrays.asList("-g", "-d", "classes");
```

Alternatively, you can use any collection class.

If you want the compiler to read source files from disk, you can ask the `StandardJavaFileManager` to translate the file name strings or `File` objects to `JavaFileObject` instances. For example,

```
StandardJavaFileManager fileManager = compiler.getStandardFileManager(null, null, null);
Iterable<JavaFileObject> fileObjects = fileManager.getJavaFileObjectsFromStrings(fileNames);
```

However, if you want the compiler to read source code from somewhere other than a disk file, you need to supply your own `JavaFileObject` subclass. Listing 10.3

shows the code for a source file object with data contained in a StringBuilder. The class extends the SimpleJavaFileObject convenience class and overrides the getCharContent method to return the content of the string builder. We'll use this class in our example program in which we dynamically produce the code for a Java class and then compile it.

The CompilationTask interface extends the Callable<Boolean> interface. You can pass it to an Executor for execution in another thread, or you can simply invoke the call method. A return value of Boolean.FALSE indicates failure.

```
Callable<Boolean> task = new JavaCompiler.CompilationTask(null, fileManager, diagnostics,
   options, null, fileObjects);
if (!task.call())
   System.out.println("Compilation failed");
```

If you simply want the compiler to produce class files on disk, you need not customize the JavaFileManager. However, our sample application will generate class files in byte arrays and later read them from memory, using a special class loader. Listing 10.4 defines a class that implements the JavaFileObject interface. Its openOutputStream method returns the ByteArrayOutputStream into which the compiler will deposit the bytecodes.

It turns out a bit tricky to tell the compiler's file manager to use these file objects. The library doesn't supply a class that implements the StandardJavaFileManager interface. Instead, you subclass the ForwardingJavaFileManager class that delegates all calls to a given file manager. In our situation, we only want to change the getJavaFileForOutput method. We achieve this with the following outline:

```
JavaFileManager fileManager = compiler.getStandardFileManager(diagnostics, null, null);
fileManager = new ForwardingJavaFileManager<JavaFileManager>(fileManager)
   {
      public JavaFileObject getJavaFileForOutput(Location location, final String className,
            Kind kind, FileObject sibling) throws IOException
      {
         return custom file object
      }
   };
```

In summary, call the run method of the JavaCompiler task if you simply want to invoke the compiler in the usual way, reading and writing disk files. You can capture the output and error messages, but you need to parse them yourself.

If you want more control over file handling or error reporting, use the CompilationTask interface instead. Its API is quite complex, but you can control every aspect of the compilation process.

Listing 10.3 compiler/StringBuilderJavaSource.java

```
package compiler;

import java.net.*;
import javax.tools.*;

/**
 * A Java source that holds the code in a string builder.
 * @version 1.00 2007-11-02
 * @author Cay Horstmann
 */
public class StringBuilderJavaSource extends SimpleJavaFileObject
{
   private StringBuilder code;

   /**
    * Constructs a new StringBuilderJavaSource.
    * @param name the name of the source file represented by this file object
    */
   public StringBuilderJavaSource(String name)
   {
      super(URI.create("string:///" + name.replace('.', '/') + Kind.SOURCE.extension),
         Kind.SOURCE);
      code = new StringBuilder();
   }

   public CharSequence getCharContent(boolean ignoreEncodingErrors)
   {
      return code;
   }

   public void append(String str)
   {
      code.append(str);
      code.append('\n');
   }
}
```

Listing 10.4 compiler/ByteArrayJavaClass.java

```
package compiler;

import java.io.*;
import java.net.*;
import javax.tools.*;
```

```
/**
 * A Java class that holds the bytecodes in a byte array.
 * @version 1.00 2007-11-02
 * @author Cay Horstmann
 */
public class ByteArrayJavaClass extends SimpleJavaFileObject
{
   private ByteArrayOutputStream stream;

   /**
    * Constructs a new ByteArrayJavaClass.
    * @param name the name of the class file represented by this file object
    */
   public ByteArrayJavaClass(String name)
   {
      super(URI.create("bytes:///" + name), Kind.CLASS);
      stream = new ByteArrayOutputStream();
   }

   public OutputStream openOutputStream() throws IOException
   {
      return stream;
   }

   public byte[] getBytes()
   {
      return stream.toByteArray();
   }
}
```

javax.tools.Tool 6

- `int run(InputStream in, OutputStream out, OutputStream err, String... arguments)`
  runs the tool with the given input, output, and error streams and the given arguments. Returns `0` for success, a nonzero value for failure.

javax.tools.JavaCompiler 6

- `StandardJavaFileManager getStandardFileManager(DiagnosticListener<? super JavaFileObject> diagnosticListener, Locale locale, Charset charset)`
  gets the standard file manager for this compiler. You can supply `null` for default error reporting, locale, and character set.

(Continues)

javax.tools.JavaCompiler 6 (Continued)

- `JavaCompiler.CompilationTask getTask(Writer out, JavaFileManager fileManager, DiagnosticListener<? super JavaFileObject> diagnosticListener, Iterable<String> options, Iterable<String> classesForAnnotationProcessing, Iterable<? extends JavaFileObject> sourceFiles)`
  gets a compilation task that, when called, will compile the given source files. See the discussion in the preceding section for details.

javax.tools.StandardJavaFileManager 6

- `Iterable<? extends JavaFileObject> getJavaFileObjectsFromStrings(Iterable<String> fileNames)`
- `Iterable<? extends JavaFileObject> getJavaFileObjectsFromFiles(Iterable<? extends File> files)`
  translates a sequence of file names or files into a sequence of `JavaFileObject` instances.

javax.tools.JavaCompiler.CompilationTask 6

- `Boolean call()`
  performs the compilation task.

javax.tools.DiagnosticCollector<S> 6

- `DiagnosticCollector()`
  constructs an empty collector.
- `List<Diagnostic<? extends S>> getDiagnostics()`
  gets the collected diagnostics.

javax.tools.Diagnostic<S> 6

- `S getSource()`
  gets the source object associated with this diagnostic.
- `Diagnostic.Kind getKind()`
  gets the type of this diagnostic—one of `ERROR`, `WARNING`, `MANDATORY_WARNING`, `NOTE`, or `OTHER`.
- `String getMessage(Locale locale)`
  gets the message describing the issue raised in this diagnostic. Pass `null` for the default locale.
- `long getLineNumber()`
- `long getColumnNumber()`
  gets the position of the issue raised in this diagnostic.

javax.tools.SimpleJavaFileObject 6

- `CharSequence getCharContent(boolean ignoreEncodingErrors)`
  override this method for a file object that represents a source file and produces the source code.
- `OutputStream openOutputStream()`
  override this method for a file object that represents a class file and produces a stream to which the bytecodes can be written.

javax.tools.ForwardingJavaFileManager<M extends JavaFileManager> 6

- `protected ForwardingJavaFileManager(M fileManager)`
  constructs a `JavaFileManager` that delegates all calls to the given file manager.
- `FileObject getFileForOutput(JavaFileManager.Location location, String className, JavaFileObject.Kind kind, FileObject sibling)`
  intercept this call if you want to substitute a file object for writing class files; `kind` is one of `SOURCE`, `CLASS`, `HTML`, or `OTHER`.

10.2.3 An Example: Dynamic Java Code Generation

In the JSP technology for dynamic web pages, you can mix HTML with snippets of Java code, such as

```
<p>The current date and time is <b><%= new java.util.Date() %></b>.</p>
```

The JSP engine dynamically compiles the Java code into a servlet. In our sample application, we use a simpler example and generate dynamic Swing code instead. The idea is that you use a GUI builder to lay out the components in a frame and specify the behavior of the components in an external file. Listing 10.5 shows a very simple example of a frame class, and Listing 10.6 shows the code for the button actions. Note that the constructor of the frame class calls an abstract method `addEventHandlers`. Our code generator will produce a subclass that implements the `addEventHandlers` method, adding an action listener for each line in the `action.properties` file. (We leave it as the proverbial exercise to the reader to extend the code generation to other event types.)

We place the subclass into a package with the name `x`, which we hope is not used anywhere else in the program. The generated code has the form

```
package x;
public class Frame extends SuperclassName {
   protected void addEventHandlers() {
      componentName1.addActionListener(new java.awt.event.ActionListener() {
         public void actionPerformed(java.awt.event.ActionEvent) {
            code for event handler1
         } } );
      // repeat for the other event handlers ...
   } }
```

The `buildSource` method in the program of Listing 10.7 builds up this code and places it into a `StringBuilderJavaSource` object. That object is passed to the Java compiler.

We use a `ForwardingJavaFileManager` with a `getJavaFileForOutput` method that constructs a `ByteArrayJavaClass` object for every class in the `x` package. These objects capture the class files generated when the `x.Frame` class is compiled. The method adds each file object to a list before returning it so that we can locate the bytecodes later. Note that compiling the `x.Frame` class produces a class file for the main class and one class file per listener class.

After compilation, we build a map that associates class names with bytecode arrays. A simple class loader (shown in Listing 10.8) loads the classes stored in this map.

We ask the class loader to load the class that we just compiled, and then we construct and display the application's frame class.

```
ClassLoader loader = new MapClassLoader(byteCodeMap);
Class<?> cl = loader.loadClass("x.Frame");
Frame frame = (JFrame) cl.newInstance();
frame.setVisible(true);
```

When you click the buttons, the background color changes in the usual way. To see that the actions are dynamically compiled, change one of the lines in `action.properties`, for example, like this:

```
yellowButton=panel.setBackground(java.awt.Color.YELLOW); yellowButton.setEnabled(false);
```

Run the program again. Now the Yellow button is disabled after you click it. Also have a look at the code directories. You will not find any source or class files for the classes in the `x` package. This example demonstrates how you can use dynamic compilation with in-memory source and class files.

Listing 10.5 buttons2/ButtonFrame.java

```java
package buttons2;
import javax.swing.*;

/**
 * @version 1.00 2007-11-02
 * @author Cay Horstmann
 */
public abstract class ButtonFrame extends JFrame
{
   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;

   protected JPanel panel;
   protected JButton yellowButton;
   protected JButton blueButton;
   protected JButton redButton;

   protected abstract void addEventHandlers();

   public ButtonFrame()
   {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      panel = new JPanel();
      add(panel);

      yellowButton = new JButton("Yellow");
      blueButton = new JButton("Blue");
      redButton = new JButton("Red");

      panel.add(yellowButton);
      panel.add(blueButton);
      panel.add(redButton);

      addEventHandlers();
   }
}
```

Listing 10.6 buttons2/action.properties

```
yellowButton=panel.setBackground(java.awt.Color.YELLOW);
blueButton=panel.setBackground(java.awt.Color.BLUE);
```

Listing 10.7 compiler/CompilerTest.java

```
package compiler;

import java.awt.*;
import java.io.*;
import java.util.*;
import java.util.List;
import javax.swing.*;
import javax.tools.*;
import javax.tools.JavaFileObject.*;

/**
 * @version 1.00 2007-10-28
 * @author Cay Horstmann
 */
public class CompilerTest
{
   public static void main(final String[] args) throws IOException, ClassNotFoundException
   {
      JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();

      final List<ByteArrayJavaClass> classFileObjects = new ArrayList<>();

      DiagnosticCollector<JavaFileObject> diagnostics = new DiagnosticCollector<>();

      JavaFileManager fileManager = compiler.getStandardFileManager(diagnostics, null, null);
      fileManager = new ForwardingJavaFileManager<JavaFileManager>(fileManager)
         {
            public JavaFileObject getJavaFileForOutput(Location location, final String className,
                  Kind kind, FileObject sibling) throws IOException
            {
               if (className.startsWith("x."))
               {
                  ByteArrayJavaClass fileObject = new ByteArrayJavaClass(className);
                  classFileObjects.add(fileObject);
                  return fileObject;
               }
               else return super.getJavaFileForOutput(location, className, kind, sibling);
            }
         };


      String frameClassName = args.length == 0 ? "buttons2.ButtonFrame" : args[0];
      JavaFileObject source = buildSource(frameClassName);
      JavaCompiler.CompilationTask task = compiler.getTask(null, fileManager, diagnostics, null,
            null, Arrays.asList(source));
```

```
      Boolean result = task.call();

      for (Diagnostic<? extends JavaFileObject> d : diagnostics.getDiagnostics())
         System.out.println(d.getKind() + ": " + d.getMessage(null));
      fileManager.close();
      if (!result)
      {
         System.out.println("Compilation failed.");
         System.exit(1);
      }

      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               try
               {
                  Map<String, byte[]> byteCodeMap = new HashMap<>();
                  for (ByteArrayJavaClass cl : classFileObjects)
                     byteCodeMap.put(cl.getName().substring(1), cl.getBytes());
                  ClassLoader loader = new MapClassLoader(byteCodeMap);
                  JFrame frame = (JFrame) loader.loadClass("x.Frame").newInstance();
                  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                  frame.setTitle("CompilerTest");
                  frame.setVisible(true);
               }
               catch (Exception ex)
               {
                  ex.printStackTrace();
               }
            }
         });
   }

   /*
    * Builds the source for the subclass that implements the addEventHandlers method.
    * @return a file object containing the source in a string builder
    */
   static JavaFileObject buildSource(String superclassName)
      throws IOException, ClassNotFoundException
   {
      StringBuilderJavaSource source = new StringBuilderJavaSource("x.Frame");
      source.append("package x;\n");
      source.append("public class Frame extends " + superclassName + " {");
      source.append("protected void addEventHandlers() {");
      final Properties props = new Properties();
      props.load(Class.forName(superclassName).getResourceAsStream("action.properties"));
```

(Continues)

Listing 10.7 *(Continued)*

```
      for (Map.Entry<Object, Object> e : props.entrySet())
      {
         String beanName = (String) e.getKey();
         String eventCode = (String) e.getValue();
         source.append(beanName + ".addActionListener(new java.awt.event.ActionListener() {");
         source.append("public void actionPerformed(java.awt.event.ActionEvent event) {");
         source.append(eventCode);
         source.append("} } );");
      }
      source.append("} }");
      return source;
   }
}
```

Listing 10.8 compiler/MapClassLoader.java

```
package compiler;

import java.util.*;

/**
 * A class loader that loads classes from a map whose keys are class names and whose values are
 * byte code arrays.
 * @version 1.00 2007-11-02
 * @author Cay Horstmann
 */
public class MapClassLoader extends ClassLoader
{
   private Map<String, byte[]> classes;

   public MapClassLoader(Map<String, byte[]> classes)
   {
      this.classes = classes;
   }

   protected Class<?> findClass(String name) throws ClassNotFoundException
   {
      byte[] classBytes = classes.get(name);
      if (classBytes == null) throw new ClassNotFoundException(name);
      Class<?> cl = defineClass(name, classBytes, 0, classBytes.length);
      if (cl == null) throw new ClassNotFoundException(name);
      return cl;
   }
}
```

10.3 Using Annotations

Annotations are tags that you insert into your source code so that some tool can process them. The tools can operate on the source level, or they can process class files into which the compiler has placed annotations.

Annotations do not change the way in which your programs are compiled. The Java compiler generates the same virtual machine instructions with or without the annotations.

To benefit from annotations, you need to select a *processing tool*. You need to use annotations that your processing tool understands, then apply the processing tool to your code.

There is a wide range of uses for annotations, and that generality can be confusing at first. Here are some uses for annotations:

- Automatic generation of auxiliary files, such as deployment descriptors or bean information classes
- Automatic generation of code for testing, logging, transaction semantics, and so on

We'll start our discussion of annotations with the basic concepts and put them to use in a concrete example: We will mark methods as event listeners for AWT components, and show you an annotation processor that analyzes the annotations and hooks up the listeners. We'll then discuss the syntax rules in detail and finish the chapter with two advanced examples for annotation processing. One of them processes source-level annotations, the other uses the Apache Bytecode Engineering Library to process class files, injecting additional bytecodes into annotated methods.

Here is an example of a simple annotation:

```
public class MyClass
{
   . . .
   @Test public void checkRandomInsertions()
}
```

The annotation `@Test` annotates the `checkRandomInsertions` method.

In Java, an annotation is used like a *modifier* and is placed before the annotated item *without a semicolon*. (A modifier is a keyword such as `public` or `static`.) The name of each annotation is preceded by an `@` symbol, similar to Javadoc comments. However, Javadoc comments occur inside `/** . . . */` delimiters, whereas annotations are part of the code.

By itself, the `@Test` annotation does not do anything. It needs a tool to be useful. For example, the JUnit 4 testing tool (available at http://junit.org) calls all methods that are labeled `@Test` when testing a class. Another tool might remove all test methods from a class file so that they are not shipped with the program after it has been tested.

Annotations can be defined to have *elements*, such as

```
@Test(timeout="10000")
```

These elements can be processed by the tools that read the annotations. Other forms of elements are possible; we'll discuss them later in this chapter.

Besides methods, you can annotate classes, fields, and local variables—an annotation can be anywhere you could put a modifier such as `public` or `static`.

Each annotation must be defined by an *annotation interface*. The methods of the interface correspond to the elements of the annotation. For example, the JUnit `Test` annotation is defined by the following interface:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test
{
   long timeout() default 0L;
   . . .
}
```

The `@interface` declaration creates an actual Java interface. Tools that process annotations receive objects that implement the annotation interface. A tool would call the `timeout` method to retrieve the `timeout` element of a particular `Test` annotation.

The `Target` and `Retention` annotations are *meta-annotations*. They annotate the `Test` annotation, marking it as an annotation that can be applied to methods only and is retained when the class file is loaded into the virtual machine. We'll discuss these in detail in Section 10.5.3, "Meta-Annotations," on p. 933.

You have now seen the basic concepts of program metadata and annotations. In the next section, we'll walk through a concrete example of annotation processing.

10.3.1 An Example: Annotating Event Handlers

One of the more boring tasks in user interface programming is the wiring of listeners to event sources. Many listeners are of the form

```
myButton.addActionListener(new
   ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         doSomething();
      }
   });
```

In this section, we'll design an annotation to avoid this drudgery. The annotation, defined in Listing 10.9, is used as follows:

```
@ActionListenerFor(source="myButton") void doSomething() { . . . }
```

The programmer no longer has to make calls to `addActionListener`. Instead, each method is simply tagged with an annotation. Listing 10.10 shows the `ButtonFrame` class from Volume I, Chapter 8, reimplemented with these annotations.

We also need to define an annotation interface. The code is in Listing 10.11.

Of course, the annotations don't do anything by themselves. They sit in the source file. The compiler places them in the class file, and the virtual machine loads them. We now need a mechanism to analyze them and install action listeners. That is the job of the `ActionListenerInstaller` class. The `ButtonFrame` constructor calls

```
ActionListenerInstaller.processAnnotations(this);
```

The static `processAnnotations` method enumerates all methods of the object it received. For each method, it gets the `ActionListenerFor` annotation object and processes it.

```
Class<?> cl = obj.getClass();
for (Method m : cl.getDeclaredMethods())
{
   ActionListenerFor a = m.getAnnotation(ActionListenerFor.class);
   if (a != null) . . .
}
```

Here, we use the `getAnnotation` method defined in the `AnnotatedElement` interface. The classes `Method`, `Constructor`, `Field`, `Class`, and `Package` implement this interface.

The name of the source field is stored in the annotation object. We retrieve it by calling the `source` method, and then look up the matching field.

```
String fieldName = a.source();
Field f = cl.getDeclaredField(fieldName);
```

This shows a limitation of our annotation. The source element must be the name of a field. It cannot be a local variable.

The remainder of the code is rather technical. For each annotated method, we construct a proxy object, implementing the `ActionListener` interface, with an `actionPerformed` method that calls the annotated method. (For more information about proxies, see Volume I, Chapter 6.) The details are not important. The key observation is that the functionality of the annotations was established by the `processAnnotations` method.

Figure 10.1 shows how annotations are handled in this example.

In this example, the annotations were processed at runtime. It is also possible to process them at the source level; a source code generator would then produce the code for adding the listeners. Alternatively, the annotations can be processed at the bytecode level; a bytecode editor could inject the calls to `addActionListener` into the frame constructor. This sounds complex, but libraries are available to make this task relatively straightforward. You can see an example in Section 10.7, "Bytecode Engineering," on p. 943.

Our example was not intended as a serious tool for user interface programmers. A utility method for adding a listener could be just as convenient for the programmer as the annotation. (In fact, the `java.beans.EventHandler` class tries to do just that. You could make the class truly useful by supplying a method that adds the event handler instead of just constructing it.)

However, this example shows the mechanics of annotating a program and of analyzing the annotations. Having seen a concrete example, you are now more prepared (we hope) for the following sections that describe the annotation syntax in complete detail.

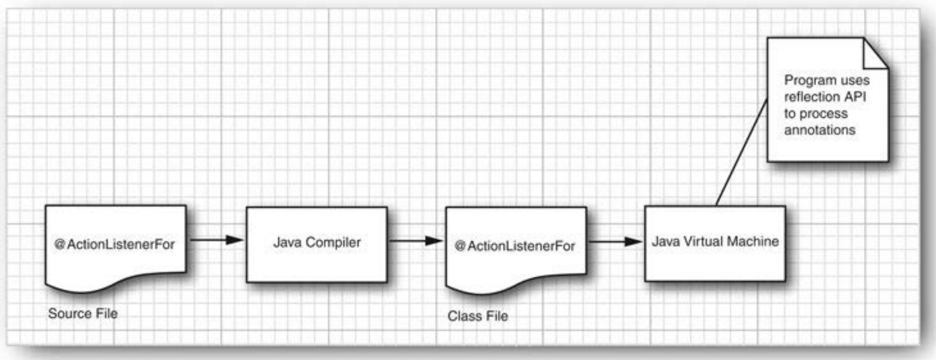

Figure 10.1 Processing annotations at runtime

Listing 10.9 runtimeAnnotations/ActionListenerInstaller.java

```
package runtimeAnnotations;

import java.awt.event.*;
import java.lang.reflect.*;

/**
 * @version 1.00 2004-08-17
 * @author Cay Horstmann
 */
public class ActionListenerInstaller
{
   /**
    * Processes all ActionListenerFor annotations in the given object.
    * @param obj an object whose methods may have ActionListenerFor annotations
    */
   public static void processAnnotations(Object obj)
   {
      try
      {
         Class<?> cl = obj.getClass();
         for (Method m : cl.getDeclaredMethods())
         {
            ActionListenerFor a = m.getAnnotation(ActionListenerFor.class);
            if (a != null)
            {
               Field f = cl.getDeclaredField(a.source());
               f.setAccessible(true);
               addListener(f.get(obj), obj, m);
            }
         }
      }
      catch (ReflectiveOperationException e)
      {
         e.printStackTrace();
      }
   }

   /**
    * Adds an action listener that calls a given method.
    * @param source the event source to which an action listener is added
    * @param param the implicit parameter of the method that the listener calls
    * @param m the method that the listener calls
    */
```

(Continues)

Listing 10.9 *(Continued)*

```
   public static void addListener(Object source, final Object param, final Method m)
         throws ReflectiveOperationException
   {
      InvocationHandler handler = new InvocationHandler()
         {
            public Object invoke(Object proxy, Method mm, Object[] args) throws Throwable
            {
               return m.invoke(param);
            }
         };

      Object listener = Proxy.newProxyInstance(null,
            new Class[] { java.awt.event.ActionListener.class }, handler);
      Method adder = source.getClass().getMethod("addActionListener", ActionListener.class);
      adder.invoke(source, listener);
   }
}
```

Listing 10.10 `buttons3/ButtonFrame.java`

```
package buttons3;

import java.awt.*;
import javax.swing.*;
import runtimeAnnotations.*;

/**
 * A frame with a button panel.
 * @version 1.00 2004-08-17
 * @author Cay Horstmann
 */
public class ButtonFrame extends JFrame
{
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 200;

   private JPanel panel;
   private JButton yellowButton;
   private JButton blueButton;
   private JButton redButton;

   public ButtonFrame()
   {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      panel = new JPanel();
      add(panel);
```

```
        yellowButton = new JButton("Yellow");
        blueButton = new JButton("Blue");
        redButton = new JButton("Red");

        panel.add(yellowButton);
        panel.add(blueButton);
        panel.add(redButton);

        ActionListenerInstaller.processAnnotations(this);
     }

     @ActionListenerFor(source = "yellowButton")
     public void yellowBackground()
     {
        panel.setBackground(Color.YELLOW);
     }

     @ActionListenerFor(source = "blueButton")
     public void blueBackground()
     {
        panel.setBackground(Color.BLUE);
     }

     @ActionListenerFor(source = "redButton")
     public void redBackground()
     {
        panel.setBackground(Color.RED);
     }
  }
```

Listing 10.11 runtimeAnnotations/ActionListenerFor.java

```
package runtimeAnnotations;

import java.lang.annotation.*;

/**
 * @version 1.00 2004-08-17
 * @author Cay Horstmann
 */

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ActionListenerFor
{
   String source();
}
```

java.lang.reflect.AnnotatedElement **5.0**

- `boolean isAnnotationPresent(Class<? extends Annotation> annotationType)`
  returns `true` if this item has an annotation of the given type.
- `<T extends Annotation> T getAnnotation(Class<T> annotationType)`
  gets the annotation of the given type, or `null` if this item has no such annotation.
- `Annotation[] getAnnotations()`
  gets all annotations present for this item, including inherited annotations. If no annotations are present, an array of length `0` is returned.
- `Annotation[] getDeclaredAnnotations()`
  gets all annotations declared for this item, excluding inherited annotations. If no annotations are present, an array of length `0` is returned.

10.4 Annotation Syntax

In this section, we cover everything you need to know about the annotation syntax.

An annotation is defined by an annotation interface:

```
modifiers @interface AnnotationName
{
   elementDeclaration1
   elementDeclaration2
   . . .
}
```

Each element declaration has the form

```
type elementName();
```

or

```
type elementName() default value;
```

For example, the following annotation has two elements, `assignedTo` and `severity`:

```
public @interface BugReport
{
   String assignedTo() default "[none]";
   int severity() = 0;
}
```

Each annotation has the format

```
@AnnotationName(elementName1=value1, elementName2=value2, . . .)
```

For example,

```
@BugReport(assignedTo="Harry", severity=10)
```

The order of the elements does not matter. The annotation

```
@BugReport(severity=10, assignedTo="Harry")
```

is identical to the preceding one.

The default value of the declaration is used if an element value is not specified. For example, consider the annotation

```
@BugReport(severity=10)
```

The value of the `assignedTo` element is the string `"[none]"`.

CAUTION: Defaults are not stored with the annotation; instead, they are dynamically computed. For example, if you change the default for the `assignedTo` element to `"[]"` and recompile the `BugReport` interface, the annotation `@BugReport(severity=10)` will use the new default, even in class files that have been compiled before the default changed.

Two special shortcuts can simplify annotations.

If no elements are specified, either because the annotation doesn't have any or because all of them use the default value, you don't need to use parentheses. For example,

```
@BugReport
```

is the same as

```
@BugReport(assignedTo="[none]", severity=0)
```

Such an annotation is called a *marker annotation*.

The other shortcut is the *single value annotation*. If an element has the special name `value` and no other element is specified, you can omit the element name and the `=` symbol. For example, had we defined the `ActionListenerFor` annotation interface of the preceding section as

```
public @interface ActionListenerFor
{
   String value();
}
```

then the annotations could be written as

```
@ActionListenerFor("yellowButton")
```

instead of

```
@ActionListenerFor(value="yellowButton")
```

All annotation interfaces implicitly extend the `java.lang.annotation.Annotation` interface. That interface is a regular interface, *not* an annotation interface. See the API notes at the end of this section for the methods provided by this interface.

You cannot extend annotation interfaces. In other words, all annotation interfaces directly extend `java.lang.annotation.Annotation`.

You never supply classes that implement annotation interfaces. Instead, the virtual machine generates proxy classes and objects when needed. For example, when requesting an `ActionListenerFor` annotation, the virtual machine carries out an operation similar to the following:

```
return Proxy.newProxyInstance(classLoader, ActionListenerFor.class,
   new
      InvocationHandler()
      {
         public Object invoke(Object proxy, Method m, Object[] args) throws Throwable
         {
            if (m.getName().equals("source")) return value of source annotation;
            . . .
         }
      });
```

The element declarations in the annotation interface are actually method declarations. The methods of an annotation interface can have no parameters and no `throws` clauses, and they cannot be generic.

The type of an annotation element is one of the following:

- A primitive type (`int`, `short`, `long`, `byte`, `char`, `double`, `float`, or `boolean`)
- `String`
- `Class` (with an optional type parameter such as `Class<? extends MyClass>`)
- An `enum` type
- An annotation type
- An array of the preceding types (an array of arrays is not a legal element type)

Here are examples of valid element declarations:

```
public @interface BugReport
{
   enum Status { UNCONFIRMED, CONFIRMED, FIXED, NOTABUG };
   boolean showStopper() default false;
   String assignedTo() default "[none]";
   Class<?> testCase() default Void.class;
```

```
    Status status() default Status.UNCONFIRMED;
    Reference ref() default @Reference(); // an annotation type
    String[] reportedBy();
}
```

Since annotations are evaluated by the compiler, all element values must be compile-time constants. For example,

```
@BugReport(showStopper=true, assignedTo="Harry", testCase=MyTestCase.class,
    status=BugReport.Status.CONFIRMED, . . .)
```

CAUTION: An annotation element can never be set to `null`. Not even a `default` of `null` is permissible. This can be rather inconvenient in practice. You will need to find other defaults, such as "" or `Void.class`.

If an element value is an array, enclose its values in braces:

```
@BugReport(. . ., reportedBy={"Harry", "Carl"})
```

You can omit the braces if the element has a single value:

```
@BugReport(. . ., reportedBy="Joe") // OK, same as {"Joe"}
```

Since an annotation element can be another annotation, you can build arbitrarily complex annotations. For example,

```
@BugReport(ref=@Reference(id="3352627"), . . .)
```

NOTE: It is an error to introduce circular dependencies in annotations. For example, `BugReport` has an element of the annotation type `Reference`, therefore `Reference` cannot have an element of type `BugReport`.

You can add annotations to the following items:

- Packages
- Classes (including `enum`)
- Interfaces (including annotation interfaces)
- Methods
- Constructors
- Instance fields (including `enum` constants)
- Local variables
- Parameter variables

However, annotations for local variables can only be processed at the source level. Class files do not describe local variables. Therefore, all local variable annotations are discarded when a class is compiled. Similarly, annotations for packages are not retained beyond the source level.

NOTE: A package is annotated in a file `package-info.java` that contains only the package statement preceded by annotations.

An item can have multiple annotations, provided they belong to different types. You cannot use the same annotation type more than once when annotating a particular item. For example,

```
@BugReport(showStopper=true, reportedBy="Joe")
@BugReport(reportedBy={"Harry", "Carl"})
void myMethod()
```

is a compile-time error. If this is a problem, design an annotation whose value is an array of simpler annotations:

```
@BugReports({
   @BugReport(showStopper=true, reportedBy="Joe"),
   @BugReport(reportedBy={"Harry", "Carl"})})
void myMethod()
```

java.lang.annotation.Annotation 5.0

- `Class<? extends Annotation> annotationType()`
  returns the `Class` object that represents the annotation interface of this annotation object. Note that calling `getClass` on an annotation object would return the actual class, not the interface.
- `boolean equals(Object other)`
  returns `true` if `other` is an object that implements the same annotation interface as this annotation object and if all elements of this object and `other` are equal.
- `int hashCode()`
  returns a hash code, compatible with the `equals` method, derived from the name of the annotation interface and the element values.
- `String toString()`
  returns a string representation that contains the annotation interface name and the element values; for example, `@BugReport(assignedTo=[none], severity=0)`.

10.5 Standard Annotations

Java SE defines a number of annotation interfaces in the `java.lang`, `java.lang.annotation`, and `javax.annotation` packages. Four of them are meta-annotations that describe the behavior of annotation interfaces. The others are regular annotations that you can use to annotate items in your source code. Table 10.2 shows these annotations. We'll discuss them in detail in the following two sections.

Table 10.2 The Standard Annotations

Annotation Interface	Applicable To	Purpose
`Deprecated`	All	Marks item as deprecated.
`SuppressWarnings`	All but packages and annotations	Suppresses warnings of the given type.
`Override`	Methods	Checks that this method overrides a superclass method.
`PostConstruct` `PreDestroy`	Methods	The marked method should be invoked immediately after construction or before removal.
`Resource`	Classes, interfaces, methods, fields	On a class or interface, marks it as a resource to be used elsewhere. On a method or field, marks it for "injection."
`Resources`	Classes, interfaces	Specifies an array of resources.
`Generated`	All	Marks an item as source code that has been generated by a tool.
`Target`	Annotations	Specifies the items to which this annotation can be applied.
`Retention`	Annotations	Specifies how long this annotation is retained.
`Documented`	Annotations	Specifies that this annotation should be included in the documentation of annotated items.
`Inherited`	Annotations	Specifies that this annotation, when applied to a class, is automatically inherited by its subclasses.

10.5.1 Annotations for Compilation

The `@Deprecated` annotation can be attached to any items for which use is no longer encouraged. The compiler will warn when you use a deprecated item. This annotation has the same role as the `@deprecated` Javadoc tag.

The `@SuppressWarnings` annotation tells the compiler to suppress warnings of a particular type, for example,

```
@SuppressWarnings("unchecked")
```

The `@Override` annotation applies only to methods. The compiler checks that a method with this annotation really overrides a method from the superclass. For example, if you declare

```
public MyClass
{
   @Override public boolean equals(MyClass other);
   . . .
}
```

then the compiler will report an error. After all, the `equals` method does *not* override the `equals` method of the `Object` class because that method has a parameter of type `Object`, not `MyClass`.

The `@Generated` annotation is intended for use by code generator tools. Any generated source code can be annotated to differentiate it from programmer-provided code. For example, a code editor can hide the generated code, or a code generator can remove older versions of generated code. Each annotation must contain a unique identifier for the code generator. A date string (in ISO 8601 format) and a comment string are optional. For example,

```
@Generated("com.horstmann.beanproperty", "2008-01-04T12:08:56.235-0700");
```

10.5.2 Annotations for Managing Resources

The `@PostConstruct` and `@PreDestroy` annotations are used in environments that control the lifecycle of objects, such as web containers and application servers. Methods tagged with these annotations should be invoked immediately after an object has been constructed or immediately before it is being removed.

The `@Resource` annotation is intended for resource injection. For example, consider a web application that accesses a database. Of course, the database access information should not be hardwired into the web application. Instead, the web container has some user interface for setting connection parameters and a JNDI name for a data source. In the web application, you can reference the data source like this:

```
@Resource(name="jdbc/mydb")
private DataSource source;
```

When an object containing this field is constructed, the container “injects” a reference to the data source.

10.5.3 Meta-Annotations

The `@Target` meta-annotation is applied to an annotation, restricting the items to which the annotation applies. For example,

```
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface BugReport
```

Table 10.3 shows all possible values. They belong to the enumerated type `ElementType`. You can specify any number of element types, enclosed in braces.

An annotation without an `@Target` restriction can be applied to any item. The compiler checks that you apply an annotation only to a permitted item. For example, if you apply `@BugReport` to a field, a compile-time error results.

The `@Retention` meta-annotation specifies how long an annotation is retained. You can specify at most one of the values in Table 10.4. The default is `RetentionPolicy.CLASS`.

In Listing 10.11 on p. 925, the `@ActionListenerFor` annotation was declared with `RetentionPolicy.RUNTIME` because we used reflection to process annotations. In the following two sections, you will see examples of processing annotations at the source and class file levels.

Table 10.3 Element Types for the `@Target` Annotation

Element Type	Annotation Applies To
`ANNOTATION_TYPE`	Annotation type declarations
`PACKAGE`	Packages
`TYPE`	Classes (including `enum`) and interfaces (including annotation types)
`METHOD`	Methods
`CONSTRUCTOR`	Constructors
`FIELD`	Fields (including `enum` constants)
`PARAMETER`	Method or constructor parameters
`LOCAL_VARIABLE`	Local variables

Table 10.4 Retention Policies for the `@Retention` Annotation

Retention Policy	Description
`SOURCE`	Annotations are not included in class files.
`CLASS`	Annotations are included in class files, but the virtual machine need not load them.
`RUNTIME`	Annotations are included in class files and loaded by the virtual machine. They are available through the reflection API.

The `@Documented` meta-annotation gives a hint to documentation tools such as Javadoc. Documented annotations should be treated just like other modifiers such as `protected` or `static` for documentation purposes. The use of other annotations is not included in the documentation. For example, suppose we declare `@ActionListenerFor` as a documented annotation:

```
@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ActionListenerFor
```

Now the documentation of each annotated method contains the annotation, as shown in Figure 10.2.

If an annotation is transient (such as `@BugReport`), you should probably not document its use.

NOTE: It is legal to apply an annotation to itself. For example, the `@Documented` annotation is itself annotated as `@Documented`. Therefore, the Javadoc documentation for annotations shows whether they are documented.

The `@Inherited` meta-annotation applies only to annotations for classes. When a class has an inherited annotation, then all of its subclasses automatically have the same annotation. This makes it easy to create annotations that work as marker interfaces, such as `Serializable`.

In fact, an annotation `@Serializable` would be more appropriate than the `Serializable` marker interface with no methods. A class is serializable because there is runtime support for reading and writing its fields, not because of any principles of object-oriented design. An annotation describes this fact better than does interface inheritance. Of course, the `Serializable` interface was created in JDK 1.1, long before annotations existed.

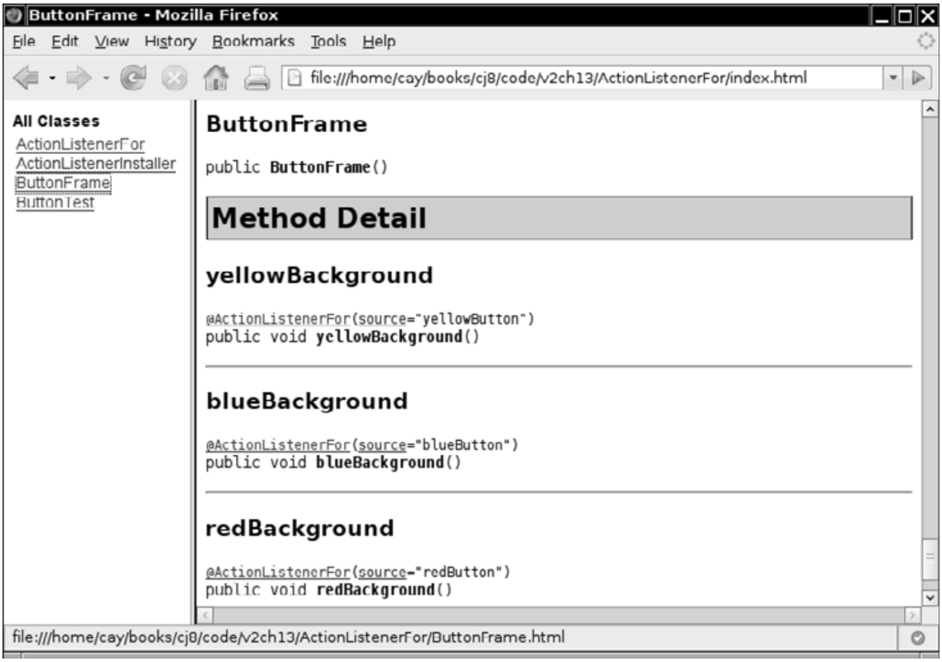

Figure 10.2 Documented annotations

Suppose you define an inherited annotation `@Persistent` to indicate that objects of a class can be saved in a database. Then the subclasses of persistent classes are automatically annotated as persistent.

```
@Inherited @interface Persistent { }
@Persistent class Employee { . . . }
class Manager extends Employee { . . . } // also @Persistent
```

When the persistence mechanism searches for objects to store in the database, it will detect both `Employee` and `Manager` objects.

10.6 Source-Level Annotation Processing

One use for annotation is the automatic generation of "side files" that contain additional information about programs. In the past, the Enterprise Edition of Java was notorious for making programmers fuss with lots of boilerplate code. Modern versions of Java EE use annotations to greatly simplify the programming model.

In this section, we demonstrate this technique with a simpler example. We write a program that automatically produces bean info classes. You tag bean properties with an annotation and then run a tool that parses the source file, analyzes the annotations, and writes out the source file of the bean info class.

Recall from Chapter 8 that a bean info class describes a bean more precisely than the automatic introspection process can. The bean info class lists all of the properties of the bean. Properties can have optional property editors. The `ChartBeanBeanInfo` class in Chapter 8 is a typical example.

To eliminate the drudgery of writing bean info classes, we supply an `@Property` annotation. You can tag either the property getter or setter, like this:

```
@Property String getTitle() { return title; }
```

or

```
@Property(editor="TitlePositionEditor")
public void setTitlePosition(int p) { titlePosition = p; }
```

Listing 10.12 contains the definition of the `@Property` annotation. Note that the annotation has a retention policy of `SOURCE`. We analyze the annotation at the source level only. It is not included in class files and not available during reflection.

Listing 10.12 `sourceAnnotations/Property.java`

```
package sourceAnnotations;
import java.lang.annotation.*;

@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Property
{
   String editor() default "";
}
```

NOTE: It would have made sense to declare the `editor` element to have type `Class`. However, the annotation processor cannot retrieve annotations of type `Class` because the meaning of a class can depend on external factors (such as the class path or class loaders). Therefore, we use a string to specify the editor class name.

To automatically generate the bean info class of a class with name *BeanClass*, we carry out the following tasks:

1. Write a source file *BeanClass*`BeanInfo.java`. Declare the *BeanClass*`BeanInfo` class to extend `SimpleBeanInfo`, and override the `getPropertyDescriptors` method.

2. For each annotated method, recover the property name by stripping off the `get` or `set` prefix and "decapitalizing" the remainder.
3. For each property, write a statement for constructing a `PropertyDescriptor`.
4. If the property has an editor, write a method call to `setPropertyEditorClass`.
5. Write the code for returning an array of all property descriptors.

For example, the annotation

```
@Property(editor="TitlePositionEditor")
public void setTitlePosition(int p) { titlePosition = p; }
```

in the `ChartBean` class is translated into

```
public class ChartBeanBeanInfo extends java.beans.SimpleBeanInfo
{
   public java.beans.PropertyDescriptor[] getProperties()
   {
      java.beans.PropertyDescriptor titlePositionDescriptor
         = new java.beans.PropertyDescriptor("titlePosition", ChartBean.class);
      titlePositionDescriptor.setPropertyEditorClass(TitlePositionEditor.class)
      . . .
      return new java.beans.PropertyDescriptor[]
      {
         titlePositionDescriptor,
         . . .
      }
   }
}
```

(The boilerplate code is printed in the lighter gray.)

All this is easy enough to do, provided we can locate all methods that have been tagged with the `@Property` annotation.

As of Java SE 6, you can add *annotation processors* to the Java compiler. (In Java SE 5, a stand-alone tool, called `apt`, was used for the same purpose.) To invoke annotation processing, run

`javac -processor` *ProcessorClassName*$_1$,*ProcessorClassName*$_2$,. . . *sourceFiles*

The compiler locates the annotations of the source files. It then selects the annotation processors that should be applied. Each annotation processor is executed in turn. If an annotation processor creates a new source file, then the process is repeated. Once a processing round yields no further source files, all source files are compiled. Figure 10.3 shows how the `@Property` annotations are processed.

We do not discuss the annotation processing API in detail, but the program in Listing 10.13 will give you a flavor of its capabilities.

An annotation processor implements the `Processor` interface, generally by extending the `AbstractProcessor` class. You need to specify which annotations your processor supports. The designers of the API themselves love annotations, so they use an annotation for this purpose:

```
@SupportedAnnotationTypes("com.horstmann.annotations.Property")
public class BeanInfoAnnotationProcessor extends AbstractProcessor
```

A processor can claim specific annotation types, wildcards such as `"com.horstmann.*“` (all annotations in the `com.horstmann` package or any subpackage), or even `"*"` (all annotations).

The `BeanInfoAnnotationProcessor` has a single public method, `process`, that is called for each file. The `process` method has two parameters: the set of annotations that is being processed in this round, and a `RoundEnv` reference that contains information about the current processing round.

In the `process` method, we iterate through all annotated methods. For each method, we get the property name by stripping off the `get`, `set`, or `is` prefix and changing the next letter to lower case. Here is the outline of the code:

```
public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv)
{
   for (TypeElement t : annotations)
   {
      Map<String, Property> props = new LinkedHashMap<String, Property>();
      for (Element e : roundEnv.getElementsAnnotatedWith(t))
      {
         props.put(property name, e.getAnnotation(Property.class));
      }
   }
   write bean info source file
   return true;
}
```

The `process` method should return `true` if it *claims* all the annotations presented to it; that is, if those annotations should not be passed on to other processors.

The code for writing the source file is straightforward—just a sequence of `out.print` statements. Note that we create the output writer as follows:

```
JavaFileObject sourceFile = processingEnv.getFiler().createSourceFile(beanClassName + "BeanInfo");
PrintWriter out = new PrintWriter(sourceFile.openWriter());
```

The `AbstractProcessor` class has a protected field `processingEnv` for accessing various processing services. The `Filer` interface is responsible for creating new files and tracking them so that they can be processed in subsequent processing rounds.

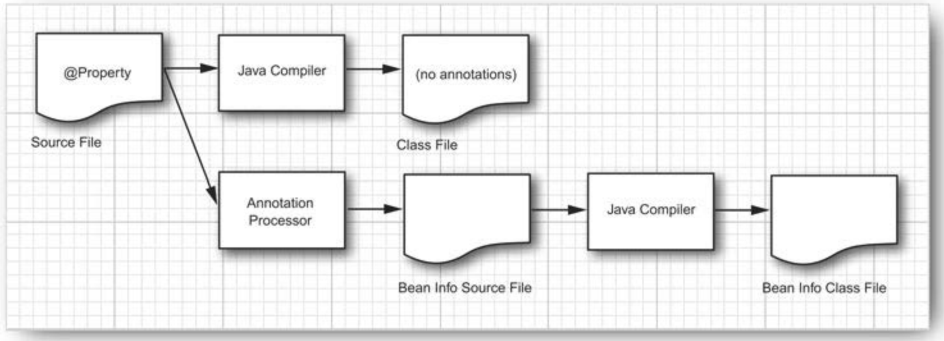

Figure 10.3 Processing source-level annotations

When an annotation processor detects an error, it uses the `Messager` to communicate with the user. For example, we issue an error message if a method has been annotated with `@Property` but its name doesn't start with `get`, `set`, or `is`:

```
if (!found) processingEnv.getMessager().printMessage(Kind.ERROR,
   "@Property must be applied to getXxx, setXxx, or isXxx method", e);
```

In the companion code for this book, we supply an annotated file, `ChartBean.java`. Compile the annotation processor:

```
javac sourceAnnotations/BeanInfoAnnotationProcessor.java
```

Then run

```
javac -processor sourceAnnotations.BeanInfoAnnotationProcessor chart/ChartBean.java
```

and have a look at the automatically generated file `ChartBeanBeanInfo.java`.

To see the annotation processing in action, add the command-line option `XprintRounds` to the `javac` command. You will get this output:

```
Round 1:
        input files: {com.horstmann.corejava.ChartBean}
        annotations: [com.horstmann.annotations.Property]
        last round: false
Round 2:
        input files: {com.horstmann.corejava.ChartBeanBeanInfo}
        annotations: []
        last round: false
Round 3:
        input files: {}
        annotations: []
        last round: true
```

This example demonstrates how tools can harvest source file annotations to produce other files. The generated files don't have to be source files. Annotation processors may choose to generate XML descriptors, property files, shell scripts, HTML documentation, and so on.

NOTE: Some people have suggested using annotations to remove an even bigger drudgery. Wouldn't it be nice if trivial getters and setters were generated automatically? For example, the annotation

```
@Property private String title;
```

could produce the methods

```
public String getTitle() { return title; }
public void setTitle(String title) { this.title = title; }
```

However, those methods need to be added to the *same class*. This requires editing a source file, not just generating another file, and is beyond the capabilities of annotation processors. It would be possible to build another tool for this purpose, but such a tool would go beyond the mission of annotations. An annotation is intended as a description *about* a code item, not a directive for adding or changing code.

Listing 10.13 sourceAnnotations/BeanInfoAnnotationProcessor.java

```java
package sourceAnnotations;

import java.beans.*;
import java.io.*;
import java.util.*;
import javax.annotation.processing.*;
import javax.lang.model.*;
import javax.lang.model.element.*;
import javax.tools.*;
import javax.tools.Diagnostic.*;

/**
 * This class is the processor that analyzes Property annotations.
 * @version 1.11 2012-01-26
 * @author Cay Horstmann
 */
@SupportedAnnotationTypes("sourceAnnotations.Property")
@SupportedSourceVersion(SourceVersion.RELEASE_7)
public class BeanInfoAnnotationProcessor extends AbstractProcessor
{
```

```
   @Override
   public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv)
   {
      for (TypeElement t : annotations)
      {
         Map<String, Property> props = new LinkedHashMap<>();
         String beanClassName = null;
         for (Element e : roundEnv.getElementsAnnotatedWith(t))
         {
            String mname = e.getSimpleName().toString();
            String[] prefixes = { "get", "set", "is" };
            boolean found = false;
            for (int i = 0; !found && i < prefixes.length; i++)
               if (mname.startsWith(prefixes[i]))
               {
                  found = true;
                  int start = prefixes[i].length();
                  String name = Introspector.decapitalize(mname.substring(start));
                  props.put(name, e.getAnnotation(Property.class));
               }

            if (!found) processingEnv.getMessager().printMessage(Kind.ERROR,
                  "@Property must be applied to getXxx, setXxx, or isXxx method", e);
            else if (beanClassName == null)
               beanClassName = ((TypeElement) e.getEnclosingElement()).getQualifiedName()
                     .toString();
         }
         try
         {
            if (beanClassName != null) writeBeanInfoFile(beanClassName, props);
         }
         catch (IOException e)
         {
            e.printStackTrace();
         }
      }
      return true;
   }

   /**
    * Writes the source file for the BeanInfo class.
    * @param beanClassName the name of the bean class
    * @param props a map of property names and their annotations
    */
   private void writeBeanInfoFile(String beanClassName, Map<String, Property> props)
      throws IOException
   {
      JavaFileObject sourceFile = processingEnv.getFiler().createSourceFile(
         beanClassName + "BeanInfo");
```

(Continues)

Listing 10.13 *(Continued)*

```
      PrintWriter out = new PrintWriter(sourceFile.openWriter());
      int i = beanClassName.lastIndexOf(".");
      if (i > 0)
      {
         out.print("package ");
         out.print(beanClassName.substring(0, i));
         out.println(";");
      }
      out.print("public class ");
      out.print(beanClassName.substring(i + 1));
      out.println("BeanInfo extends java.beans.SimpleBeanInfo");
      out.println("{");
      out.println("   public java.beans.PropertyDescriptor[] getPropertyDescriptors()");
      out.println("   {");
      out.println("      try");
      out.println("      {");
      for (Map.Entry<String, Property> e : props.entrySet())
      {
         out.print("         java.beans.PropertyDescriptor ");
         out.print(e.getKey());
         out.println("Descriptor");
         out.print("            = new java.beans.PropertyDescriptor(\"");
         out.print(e.getKey());
         out.print("\", ");
         out.print(beanClassName);
         out.println(".class);");
         String ed = e.getValue().editor().toString();
         if (!ed.equals(""))
         {
            out.print("         ");
            out.print(e.getKey());
            out.print("Descriptor.setPropertyEditorClass(");
            out.print(ed);
            out.println(".class);");
         }
      }
      out.println("         return new java.beans.PropertyDescriptor[]");
      out.print("         {");
      boolean first = true;
      for (String p : props.keySet())
      {
         if (first) first = false;
         else out.print(",");
         out.println();
         out.print("            ");
         out.print(p);
         out.print("Descriptor");
      }
```

```
      out.println();
      out.println("         };");
      out.println("      }");
      out.println("      catch (java.beans.IntrospectionException e)");
      out.println("      {");
      out.println("         e.printStackTrace();");
      out.println("         return null;");
      out.println("      }");
      out.println("   }");
      out.println("}");
      out.close();
   }
}
```

10.7 Bytecode Engineering

You have seen how annotations can be processed at runtime or at the source code level. There is a third possibility: processing at the bytecode level. Unless annotations are removed at the source level, they are present in the class files. The class file format is documented (see http://docs.oracle.com/javase/specs/jvms/se7/html). The format is rather complex, and it would be challenging to process class files without special libraries. One such library is the Bytecode Engineering Library (BCEL), available at http://jakarta.apache.org/bcel.

In this section, we use BCEL to add logging messages to annotated methods. If a method is annotated with

```
@LogEntry(logger=loggerName)
```

then we add the bytecodes for the following statement at the beginning of the method:

```
Logger.getLogger(loggerName).entering(className, methodName);
```

For example, if you annotate the `hashCode` method of the `Item` class as

```
@LogEntry(logger="global") public int hashCode()
```

then a message similar to the following is printed whenever the method is called:

```
Aug 17, 2004 9:32:59 PM Item hashCode
FINER: ENTRY
```

To achieve this, we do the following:

1. Load the bytecodes in the class file.
2. Locate all methods.
3. For each method, check whether it has a `LogEntry` annotation.

4. If it does, add the bytecodes for the following instructions at the beginning of the method:

```
ldc loggerName
invokestatic java/util/logging/Logger.getLogger:(Ljava/lang/String;)Ljava/util/logging/Logger;
ldc className
ldc methodName
invokevirtual java/util/logging/Logger.entering:(Ljava/lang/String;Ljava/lang/String;)V
```

Inserting these bytecodes sounds tricky, but BCEL makes it fairly straightforward. We don't describe the process of analyzing and inserting bytecodes in detail. The important point is that the program in Listing 10.14 edits a class file and inserts a logging call at the beginning of the methods annotated with the `LogEntry` annotation.

NOTE: If you are interested in the details of bytecode engineering, read through the BCEL manual at http://jakarta.apache.org/bcel/manual.html.

You'll need version 6.0 or later of the BCEL library to compile and run the `EntryLogger` program. (As this chapter was written, that version was still a work in progress. If it isn't finished when you read this, check out the trunk from the Subversion repository.)

For example, here is how you add the logging instructions to `Item.java` in Listing 10.15:

```
javac set/Item.java
javac -classpath .:bcel-version.jar bytecodeAnnotations/EntryLogger.java
java -classpath .:bcel-version.jar bytecodeAnnotations.EntryLogger set.Item
```

Try running

```
javap -c set.Item
```

before and after modifying the `Item` class file. You can see the inserted instructions at the beginning of the `hashCode`, `equals`, and `compareTo` methods.

```
public int hashCode();
  Code:
   0:   ldc     #85; //String global
   2:   invokestatic    #80;
      //Method java/util/logging/Logger.getLogger:(Ljava/lang/String;)Ljava/util/logging/Logger;
   5:   ldc     #86; //String Item
   7:   ldc     #88; //String hashCode
   9:   invokevirtual   #84;
      //Method java/util/logging/Logger.entering:(Ljava/lang/String;Ljava/lang/String;)V
   12:  bipush  13
   14:  aload_0
```

```
15: getfield        #2; //Field description:Ljava/lang/String;
18: invokevirtual   #15; //Method java/lang/String.hashCode:()I
21: imul
22: bipush  17
24: aload_0
25: getfield        #3; //Field partNumber:I
28: imul
29: iadd
30: ireturn
```

The `SetTest` program in Listing 10.16 inserts `Item` objects into a hash set. When you run it with the modified class file, you will see the logging messages.

```
Aug 18, 2004 10:57:59 AM Item hashCode
FINER: ENTRY
Aug 18, 2004 10:57:59 AM Item hashCode
FINER: ENTRY
Aug 18, 2004 10:57:59 AM Item hashCode
FINER: ENTRY
Aug 18, 2004 10:57:59 AM Item equals
FINER: ENTRY
[[description=Toaster, partNumber=1729], [description=Microwave, partNumber=4104]]
```

Note the call to `equals` when we insert the same item twice.

This example shows the power of bytecode engineering. Annotations are used to add directives to a program, and a bytecode editing tool picks up the directives and modifies the virtual machine instructions.

Listing 10.14 bytecodeAnnotations/EntryLogger.java

```java
package bytecodeAnnotations;

import java.io.*;
import org.apache.bcel.*;
import org.apache.bcel.classfile.*;
import org.apache.bcel.generic.*;

/**
 * Adds "entering" logs to all methods of a class that have the LogEntry annotation.
 * @version 1.10 2007-10-27
 * @author Cay Horstmann
 */
public class EntryLogger
{
   private ClassGen cg;
   private ConstantPoolGen cpg;
```

(Continues)

Listing 10.14 *(Continued)*

```java
   /**
    * Adds entry logging code to the given class.
    * @param args the name of the class file to patch
    */
   public static void main(String[] args)
   {
      try
      {
         if (args.length == 0)
            System.out.println("USAGE: java bytecodeAnnotations.EntryLogger classname");
         else
         {
            JavaClass jc = Repository.lookupClass(args[0]);
            ClassGen cg = new ClassGen(jc);
            EntryLogger el = new EntryLogger(cg);
            el.convert();
            String f = Repository.lookupClassFile(cg.getClassName()).getPath();
            System.out.println("Dumping " + f);
            cg.getJavaClass().dump(f);
         }
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }

   /**
    * Constructs an EntryLogger that inserts logging into annotated methods of a given class.
    * @param cg the class
    */
   public EntryLogger(ClassGen cg)
   {
      this.cg = cg;
      cpg = cg.getConstantPool();
   }

   /**
    * converts the class by inserting the logging calls.
    */
   public void convert() throws IOException
   {
      for (Method m : cg.getMethods())
      {
         AnnotationEntry[] annotations = m.getAnnotationEntries();
```

```
         for (AnnotationEntry a : annotations)
         {
            if (a.getAnnotationType().equals("LbytecodeAnnotations/LogEntry;"))
            {
               for (ElementValuePair p : a.getElementValuePairs())
               {
                  if (p.getNameString().equals("logger"))
                  {
                     String loggerName = p.getValue().stringifyValue();
                     cg.replaceMethod(m, insertLogEntry(m, loggerName));
                  }
               }
            }
         }
      }
   }

   /**
    * Adds an "entering" call to the beginning of a method.
    * @param m the method
    * @param loggerName the name of the logger to call
    */
   private Method insertLogEntry(Method m, String loggerName)
   {
      MethodGen mg = new MethodGen(m, cg.getClassName(), cpg);
      String className = cg.getClassName();
      String methodName = mg.getMethod().getName();
      System.out.printf("Adding logging instructions to %s.%s%n", className, methodName);

      int getLoggerIndex = cpg.addMethodref("java.util.logging.Logger", "getLogger",
            "(Ljava/lang/String;)Ljava/util/logging/Logger;");
      int enteringIndex = cpg.addMethodref("java.util.logging.Logger", "entering",
            "(Ljava/lang/String;Ljava/lang/String;)V");

      InstructionList il = mg.getInstructionList();
      InstructionList patch = new InstructionList();
      patch.append(new PUSH(cpg, loggerName));
      patch.append(new INVOKESTATIC(getLoggerIndex));
      patch.append(new PUSH(cpg, className));
      patch.append(new PUSH(cpg, methodName));
      patch.append(new INVOKEVIRTUAL(enteringIndex));
      InstructionHandle[] ihs = il.getInstructionHandles();
      il.insert(ihs[0], patch);

      mg.setMaxStack();
      return mg.getMethod();
   }
}
```

Listing 10.15 `set/Item.java`

```
package set;

import java.util.*;
import bytecodeAnnotations.*;

/**
 * An item with a description and a part number.
 * @version 1.01 2012-01-26
 * @author Cay Horstmann
 */
public class Item
{
   private String description;
   private int partNumber;

   /**
    * Constructs an item.
    * @param aDescription the item's description
    * @param aPartNumber the item's part number
    */
   public Item(String aDescription, int aPartNumber)
   {
      description = aDescription;
      partNumber = aPartNumber;
   }

   /**
    * Gets the description of this item.
    * @return the description
    */
   public String getDescription()
   {
      return description;
   }

   public String toString()
   {
      return "[description=" + description + ", partNumber=" + partNumber + "]";
   }

   @LogEntry(logger = "global")
   public boolean equals(Object otherObject)
   {
      if (this == otherObject) return true;
      if (otherObject == null) return false;
      if (getClass() != otherObject.getClass()) return false;
```

```
      Item other = (Item) otherObject;
      return Objects.equals(description, other.description) && partNumber == other.partNumber;
   }

   @LogEntry(logger = "global")
   public int hashCode()
   {
      return Objects.hash(description, partNumber);
   }
}
```

Listing 10.16 set/SetTest.java

```
package set;

import java.util.*;
import java.util.logging.*;

/**
 * @version 1.02 2012-01-26
 * @author Cay Horstmann
 */
public class SetTest
{
   public static void main(String[] args)
   {
      Logger.getLogger(Logger.GLOBAL_LOGGER_NAME).setLevel(Level.FINEST);
      Handler handler = new ConsoleHandler();
      handler.setLevel(Level.FINEST);
      Logger.getLogger(Logger.GLOBAL_LOGGER_NAME).addHandler(handler);

      Set<Item> parts = new HashSet<>();
      parts.add(new Item("Toaster", 1279));
      parts.add(new Item("Microwave", 4104));
      parts.add(new Item("Toaster", 1279));
      System.out.println(parts);
   }
}
```

10.7.1 Modifying Bytecodes at Load Time

In the preceding section, you saw a tool that edits class files. However, it can be cumbersome to add yet another tool into the build process. An attractive alternative is to defer the bytecode engineering until *load time*, when the class loader loads the class.

Before Java SE 5.0, you had to write a custom class loader to achieve this task. Now, the *instrumentation API* has a hook for installing a bytecode transformer. The transformer must be installed before the `main` method of the program is called. You can meet this requirement by defining an *agent*, a library that is loaded to monitor a program in some way. The agent code can carry out initializations in a `premain` method.

Here are the steps required to build an agent:

1. Implement a class with a method

   ```
   public static void premain(String arg, Instrumentation instr)
   ```

   This method is called when the agent is loaded. The agent can get a single command-line argument, which is passed in the `arg` parameter. The `instr` parameter can be used to install various hooks.

2. Make a manifest file `EntryLoggingAgent.mf` that sets the `Premain-Class` attribute, for example:

   ```
   Premain-Class: bytecodeAnnotations.EntryLoggingAgent
   ```

3. Package the agent code and the manifest into a JAR file, for example:

   ```
   javac -classpath .:bcel-version.jar bytecodeAnnotations.EntryLoggingAgent
   jar cvfm EntryLoggingAgent.jar EntryLoggingAgent.mf bytecodeAnnotations/Entry*.class
   ```

To launch a Java program together with the agent, use the following command-line options:

```
java -javaagent:AgentJARFile=agentArgument . . .
```

For example, to run the `SetTest` program with the entry logging agent, call

```
javac SetTest.java
java -javaagent:EntryLoggingAgent.jar=set.Item -classpath .:bcel-version.jar set.SetTest
```

The `Item` argument is the name of the class that the agent should modify.

Listing 10.17 shows the agent code. The agent installs a class file transformer. The transformer first checks whether the class name matches the agent argument. If so, it uses the `EntryLogger` class from the preceding section to modify the bytecodes. However, the modified bytecodes are not saved to a file. Instead, the transformer returns them for loading into the virtual machine (see Figure 10.4). In other words, this technique carries out “just in time” modification of the bytecodes.

Listing 10.17 bytecodeAnnotations/EntryLoggingAgent.java

```
package bytecodeAnnotations;

import java.lang.instrument.*;
import java.io.*;
import java.security.*;
import org.apache.bcel.classfile.*;
import org.apache.bcel.generic.*;

/**
 * @version 1.00 2004-08-17
 * @author Cay Horstmann
 */
public class EntryLoggingAgent
{
   public static void premain(final String arg, Instrumentation instr)
   {
      instr.addTransformer(new ClassFileTransformer()
         {
            public byte[] transform(ClassLoader loader, String className, Class<?> cl,
                  ProtectionDomain pd, byte[] data)
            {
               if (!className.equals(arg)) return null;
               try
               {
                  ClassParser parser = new ClassParser(new ByteArrayInputStream(data), className
                        + ".java");
                  JavaClass jc = parser.parse();
                  ClassGen cg = new ClassGen(jc);
                  EntryLogger el = new EntryLogger(cg);
                  el.convert();
                  return cg.getJavaClass().getBytes();
               }
               catch (Exception e)
               {
                  e.printStackTrace();
                  return null;
               }
            }
         });
   }
}
```

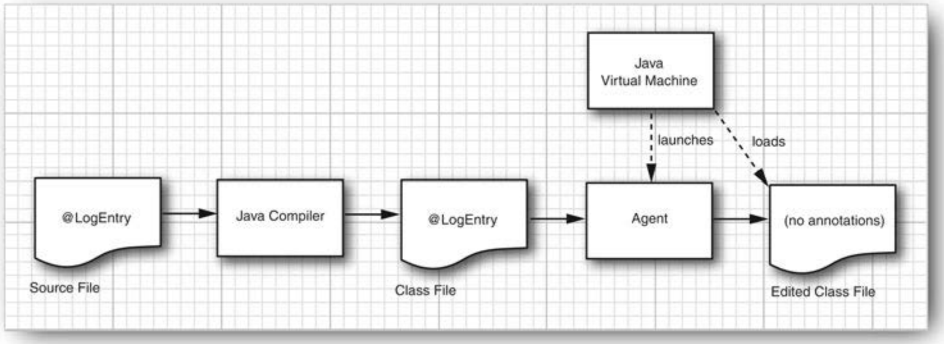

Figure 10.4 Modifying classes at load time

In this chapter, you have learned how to

- Add annotations to Java programs
- Design your own annotation interfaces
- Implement tools that make use of the annotations

You have seen three technologies for processing code: scripting, compiling Java programs, and processing annotations. The first two were quite straightforward. On the other hand, building annotation tools is undeniably complex and not something that most developers will need to tackle. This chapter gave you the background for understanding the inner workings of the annotation tools you will encounter, and perhaps piqued your interest in developing your own tools.

In the next chapter, you will learn about the RMI mechanism, a distributed object model for Java programs.

Index

Symbols and Numbers

E

H

J

O

W

Z