Mercer University

Advanced Java Programming

SSE 554 Spring 2010 – Project 3

TUYEN DO – TUNG NGUYEN - NGOC DONG

4/26/2010

Table of Contents

1 Introduction

This Team Project 3 serves as a continuation effort of coding in Java above and beyond the basic described in Core Java Volume I. The content of this report covers many advanced features described in Core Java Volume II and some other resources on internets. Topics covered are Streams and Files, XML, Database Programming, Internationaization, Advanced Swing, Advanced AWT, JavaBeans Components, Distributed Objects, and Scripting-Compiling-and Annotation Processing.

Based on our interests, objective of the project and time constrain, this paper covers as much as possible all concepts and standard features that help to accomplish our goal in providing the understanding of selected topics. But there is no way to cevered all topics in details and therefore some of the topics are broadly discussed and refered to references when necessary. The report is structured simply by two main sections for each topic: basic description, instruction, and standard followed by demonstration. Basic concepts, techniques, standards on the subject may be directly excerpted from the sources listed in references where appropriate.

More detailed discussion for particular topic that can not be found in this paper can be found in Core Java Volume II-Advances features, 8th Edition [1] if preferred.

2 Streams and Files

Streams

In the Java API, an object from which we can read a sequence of bytes is called an *input stream*. An object to which we can write a sequence of bytes is called an output stream. These sources and destinations of byte sequences can be—and often are—files, but they can also be network connections and even blocks of memory. The abstract classes *InputStream* and *OutputStream* form the basis for a hierarchy of input/output (I/O) classes. Because byte-oriented streams are inconvenient for processing information stored in Unicode (recall that Unicode uses multiple bytes per character), there is a separate hierarchy of classes for processing Unicode characters that inherit from the abstract *Reader* and *Writer* classes. These classes have read and write operations that are based on two-byte Unicode code units rather than on single-byte characters.

I/O Streams

An *I/O Stream* represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.

Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects. Some streams simply pass on data; others manipulate and transform the data in useful ways. No matter how they work internally, all streams present the same simple model to programs that use them: A stream is a sequence of data. A program uses an *input stream* to read data from a source, one item at a time:

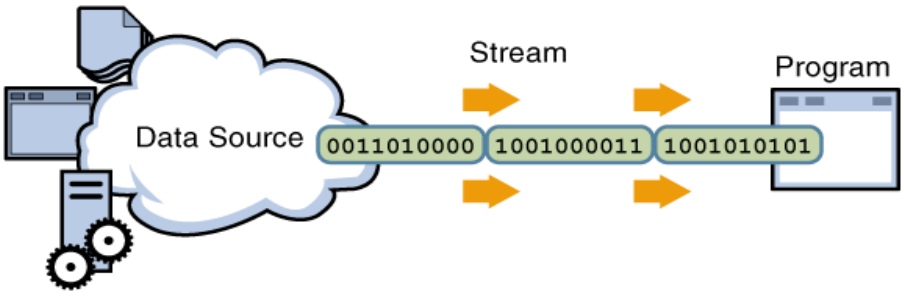


Figure 1: Reading information into a program.

A program uses an *output stream* to write data to a destination, one item at time:

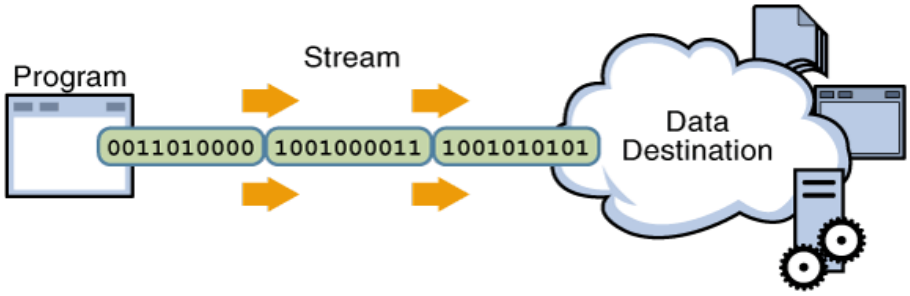


Figure 2: Writing information from a program.

We'll see streams that can handle all kinds of data, from primitive values to advanced objects. The data source and data destination from figure 2 above can be anything that holds, generates, or consumes data. Obviously this includes disk files, but a source or destination can also be another program, a peripheral device, a network socket, or an array.

2.1 Overview of Stream and Files

We will use the most basic kind of streams, byte streams, to demonstrate the common operations of Stream I/O.

Byte Streams

Programs use *byte streams* to perform input and output of 8-bit bytes. All byte stream classes are descended from *InputStream* and *OutputStream*.

There are many byte stream classes. To demonstrate how byte streams work, we'll focus on the file I/O byte streams, *FileInputStream* and *FileOutputStream*. Other kinds of byte streams are used in much the same way; they differ mainly in the way they are constructed.

Using Byte Streams

We'll explore FileInputStream and FileOutputStream by examining an example program named CopyBytes, which uses byte streams to copy xanadu.txt, one byte at a time.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
```

```
import java.io.IOException;
public class CopyBytes {
    public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("xanadu.txt");
            out = new FileOutputStream("outagain.txt");
            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

CopyBytes spends most of its time in a simple loop that reads the input stream and writes the output stream, one byte at a time, as shown in the following figure.

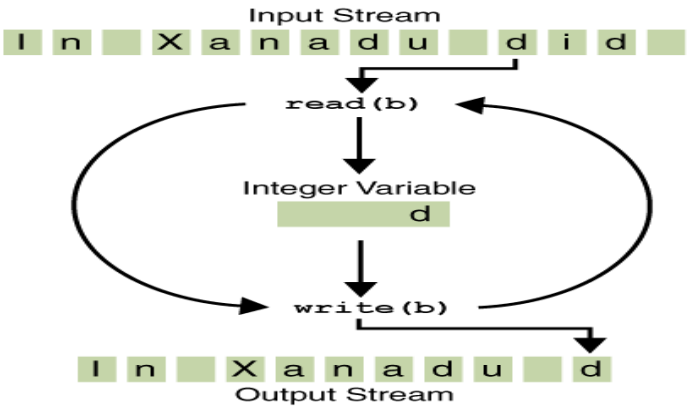


Figure 3: Simple byte stream input and output.

Notice that *read()* returns an int value. If the input is a stream of bytes, why doesn't read() return a byte value? Using a int as a return type allows *read()* to use -1 to indicate that it has reached the end of the stream.

Always Close Streams

Closing a stream when it's no longer needed is very important — so important that **CopyBytes** uses a finally block to guarantee that both streams will be closed even if an error occurs. This practice helps avoid serious resource leaks.

One possible error is that **CopyBytes** was unable to open one or both files. When that happens, the stream variable corresponding to the file never changes from its initial null value. That's why CopyBytes makes sure that each stream variable contains an object reference before invoking close.

When Not to Use Byte Streams

CopyBytes seems like a normal program, but it actually represents a kind of low-level I/O that you should avoid. Since xanadu.txt contains character data, the best approach is to use character streams, as discussed in the next section. There are also streams for more complicated data types. Byte streams should only be used for the most primitive I/O.

Character Streams

The Java platform stores character values using Unicode conventions. Character stream I/O automatically translates this internal format to and from the local character set. In Western locales, the local character set is usually an 8-bit superset of ASCII.

For most applications, I/O with character streams is no more complicated than I/O with byte streams. Input and output done with stream classes automatically translates to and from the local character set. A program that uses character streams in place of byte streams automatically adapts to the local character set and is ready for internationalization — all without extra effort by the programmer.

If internationalization isn't a priority, you can simply use the character stream classes without paying much attention to character set issues. Later, if internationalization becomes a priority, your program can be adapted without extensive recoding. Using Character Streams

All character stream classes are descended from **Reader** and **Writer**. As with byte streams, there are character stream classes that specialize in file I/O: **FileReader** and **FileWriter**. The **CopyCharacters** example illustrates these classes.

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

public class CopyCharacters {
    public static void main(String[] args) throws IOException {
        FileReader inputStream = null;
        FileWriter outputStream = null;
        try {
            inputStream = new FileReader("xanadu.txt");
            outputStream = new FileWriter("characteroutput.txt");

            int c;
            while ((c = inputStream.read()) != -1) {
                outputStream.write(c);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
```

```
                outputStream.close();
            }
        }
    }
}
```

CopyCharacters is very similar to CopyBytes. The most important difference is that CopyCharacters uses FileReader and FileWriter for input and output in place of FileInputStream and FileOutputStream. Notice that both CopyBytes and CopyCharacters use an int variable to read to and write from. However, in CopyCharacters, the int variable holds a character value in its last 16 bits; in CopyBytes, the int variable holds a byte value in its last 8 bits.

Character Streams that Use Byte Streams

Character streams are often "wrappers" for byte streams. The character stream uses the byte stream to perform the physical I/O, while the character stream handles translation between characters and bytes. FileReader, for example, uses FileInputStream, while FileWriter uses FileOutputStream.

There are two general-purpose byte-to-character "bridge" streams: **InputStreamReader** and **OutputStreamWriter**. Use them to create character streams when there are no prepackaged character stream classes that meet your needs. The **sockets lesson** in the **networking trail** shows how to create character streams from the byte streams provided by socket classes.

Line-Oriented I/O

Character I/O usually occurs in bigger units than single characters. One common unit is the line: a string of characters with a line terminator at the end. A line terminator can be a carriage-return/line-feed sequence ("\r\n"), a single carriage-return ("\r"), or a single line-feed ("\n"). Supporting all possible line terminators allows programs to read text files created on any of the widely used operating systems.

Let's modify the CopyCharacters example to use line-oriented I/O. To do this, we have to use two classes we haven't seen before, **BufferedReader** and **PrintWriter**. We'll explore these classes in greater depth in **Buffered I/O** and **Formatting**. Right now, we're just interested in their support for line-oriented I/O.

The **CopyLines** example invokes BufferedReader.readLine and PrintWriter.println to do input and output one line at a time.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;

public class CopyLines {
    public static void main(String[] args) throws IOException {
        BufferedReader inputStream = null;
        PrintWriter outputStream = null;
        try {
            inputStream =
                new BufferedReader(new FileReader("xanadu.txt"));
            outputStream =
                new PrintWriter(new FileWriter("characteroutput.txt"));

            String l;
            while ((l = inputStream.readLine()) != null) {
                outputStream.println(l);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

Invoking readLine returns a line of text with the line. CopyLines outputs each line using println, which appends the line terminator for the current operating system. This might not be the same line terminator that was used in the input file.

The Complete Stream Zoo

Unlike C, which gets by just fine with a single type FILE*, Java has a whole zoo of more than 60 (!) different stream types (see Figures 4 and 5).

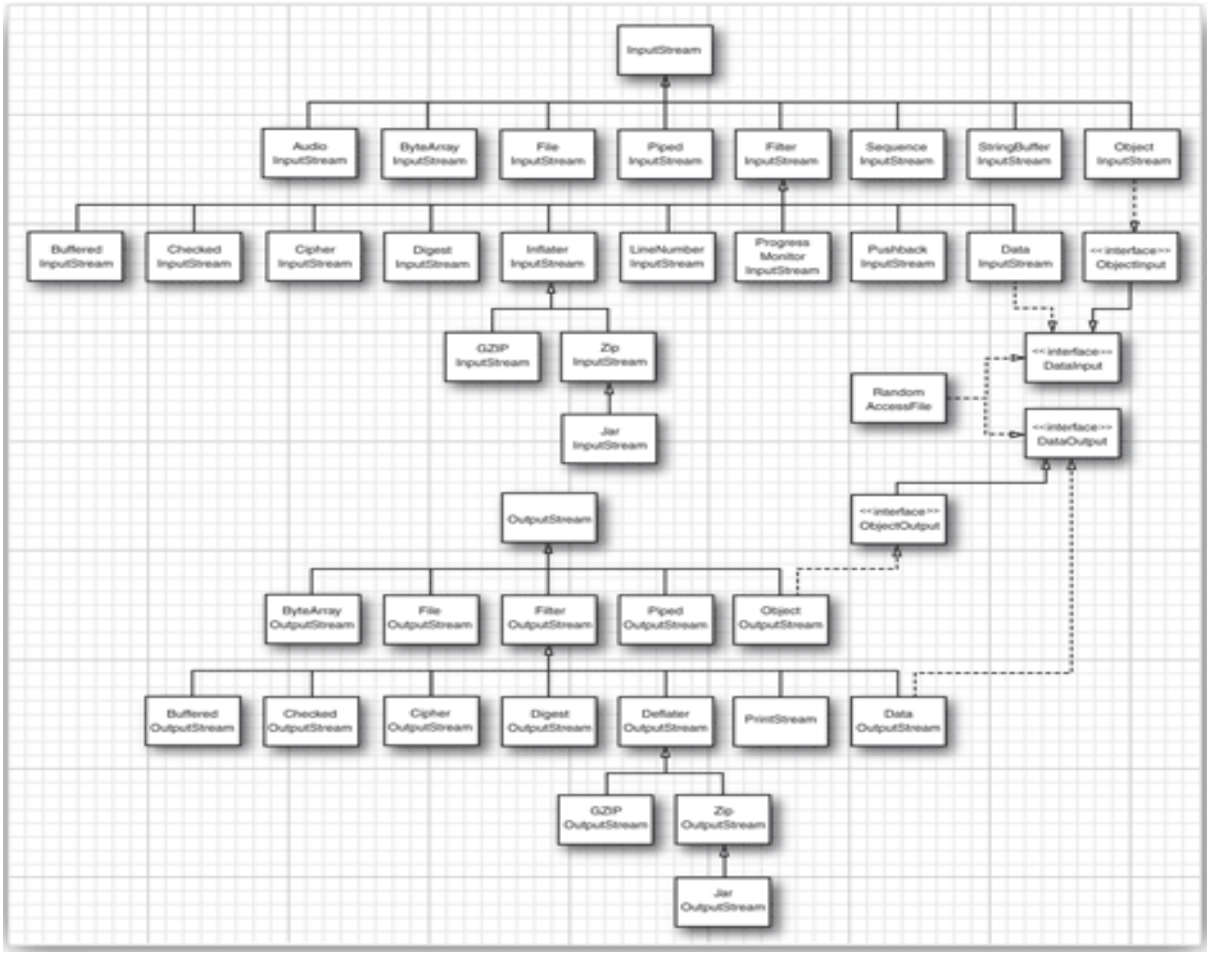


Figure 4. Input and output stream hierarchy.

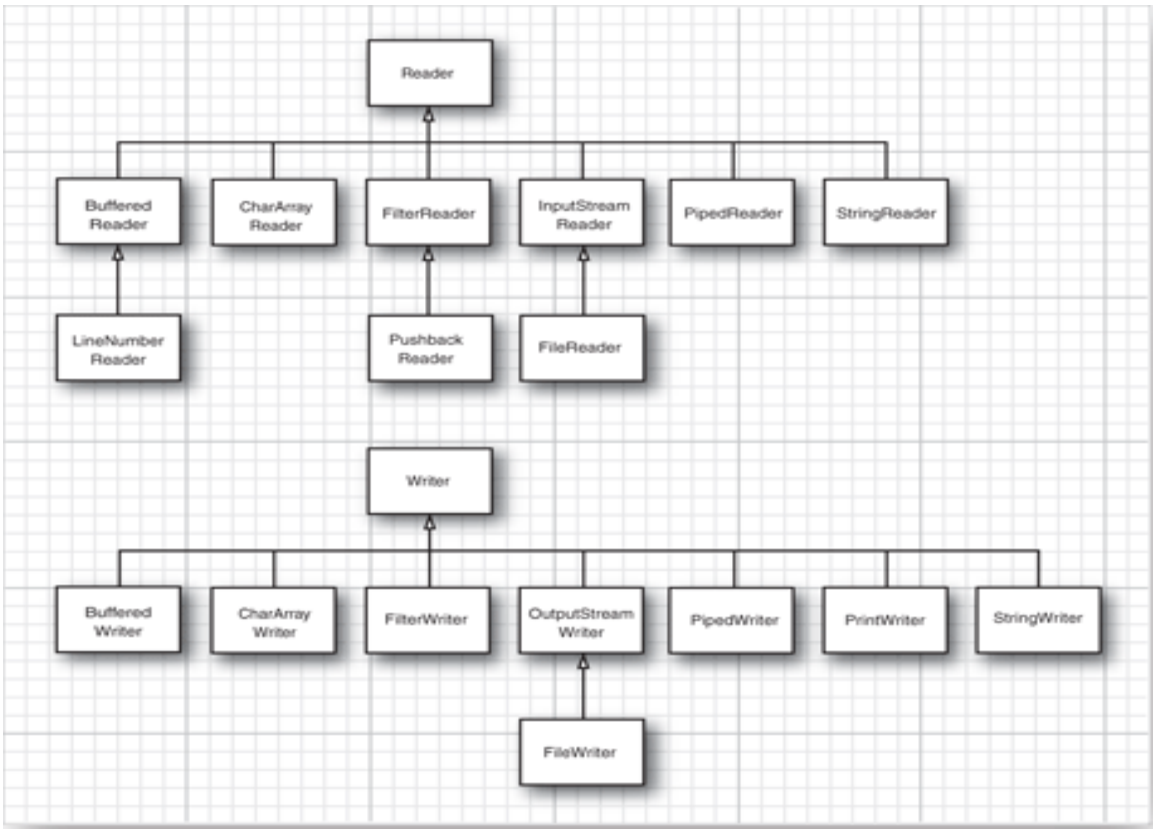


Figure 5. Reader and writer hierarchy.

Let us divide the animals in the stream class zoo by how they are used. There are separate hierarchies for classes that process bytes and characters. As you saw, the `InputStream` and `OutputStream` classes let you read and write individual bytes and arrays of bytes. These classes form the basis of the hiearchy shown in Figure 4. To read and write strings and numbers, you need more capable subclasses. For example, `DataInputStream` and `DataOutputStream` let you read and write all the primitive Java types in binary format. Finally, there are streams that do useful stuff; for example, the `ZipInputStream` and `ZipOutputStream` that let you read and write files in the familiar ZIP compression format.

For Unicode text, on the other hand, you use subclasses of the abstract classes `Reader` and `Writer` (see Figure 5). The basic methods of the `Reader` and `Writer` classes are similar to the ones for `InputStream` and `OutputStream`.

```
abstract int read()
```

```
abstract void write(int c)
```

The `read` method returns either a Unicode code unit (as an integer between 0 and 65535) or -1 when you have reached the end of the file. The `write` method is called with a Unicode code unit.

Java SE 5.0 introduced four additional interfaces: `Closeable`, `Flushable`, `Readable`, and `Appendable` (see Figure 6). The first two interfaces are very simple, with methods

```
void close() throws IOException
```

and

```
void flush()
```

respectively. The classes `InputStream`, `OutputStream`, `Reader`, and `Writer` all implement the `Closeable` interface. `OutputStream` and `Writer` implement the `Flushable` interface.

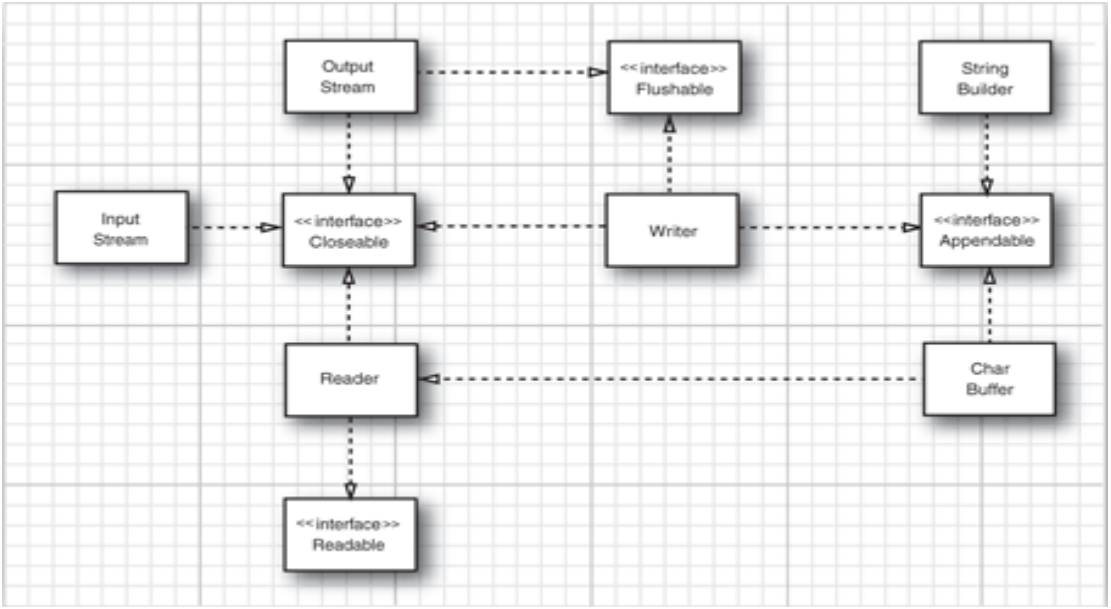


Figure 6. The Closeable, Flushable, Readable, and Appendable interfaces.

Combining Stream Filters

FileInputStream and FileOutputStream give you input and output streams attached to a disk file. You give the file name or full path name of the file in the constructor. For example,

```
FileInputStream fin = new FileInputStream("employee.dat");
```

looks in the user directory for a file named "employee.dat".

Like the abstract InputStream and OutputStream classes, these classes support only reading and writing on the byte level. That is, we can only read bytes and byte arrays from the object fin.

```
byte b = (byte) fin.read();
```

As you will see in the next section, if we just had a DataInputStream, then we could read numeric types:

```
DataInputStream din = . . .;

double s = din.readDouble();
```

But just as the FileInputStream has no methods to read numeric types, the DataInputStream has no method to get data from a file.

Java uses a clever mechanism to separate two kinds of responsibilities. Some streams (such as the FileInputStream and the input stream returned by the openStream method of the URL class) can retrieve bytes from files and other more exotic locations. Other streams (such as the DataInputStream and the PrintWriter) can assemble bytes into more useful data types. The Java programmer has to combine the two. For example, to be able to read numbers from a file, first create a FileInputStream and then pass it to the constructor of a DataInputStream.

```
FileInputStream fin = new FileInputStream("employee.dat");

DataInputStream din = new DataInputStream(fin);

double s = din.readDouble();
```

If you look at Figure 4 again, you can see the classes FilterInputStream and FilterOutputStream. The subclasses of these files are used to add capabilities to raw byte streams.

You can add multiple capabilities by nesting the filters. For example, by default, streams are not buffered. That is, every call to read asks the operating system to dole out yet another byte. It is more efficient to request blocks of data instead and put them in a buffer. If you want buffering

and the data input methods for a file, you need to use the following rather monstrous sequence of constructors:

```
DataInputStream din = new DataInputStream(
   new BufferedInputStream(
   new FileInputStream("employee.dat")));
```

Notice that we put the DataInputStream last in the chain of constructors because we want to use the DataInputStream methods, and we want them to use the buffered read method.

Sometimes you'll need to keep track of the intermediate streams when chaining them together. For example, when reading input, you often need to peek at the next byte to see if it is the value that you expect. Java provides the PushbackInputStream for this purpose.

```
PushbackInputStream pbin = new PushbackInputStream(
   new BufferedInputStream(
   new FileInputStream("employee.dat")));
```

Now you can speculatively read the next byte

```
int b = pbin.read();
```

and throw it back if it isn't what you wanted.

```
if (b != '<') pbin.unread(b);
```

But reading and unreading are the only methods that apply to the pushback input stream. If you want to look ahead and also read numbers, then you need both a pushback input stream and a data input stream reference.

```
DataInputStream din = new DataInputStream(
   pbin = new PushbackInputStream(
      new BufferedInputStream(
```

```
        new FileInputStream("employee.dat"))));
```

Of course, in the stream libraries of other programming languages, niceties such as buffering and lookahead are automatically taken care of, so it is a bit of a hassle in Java that one has to resort to combining stream filters in these cases. But the ability to mix and match filter classes to construct truly useful sequences of streams does give you an immense amount of flexibility. For example, you can read numbers from a compressed ZIP file by using the following sequence of streams (see Figure 7):

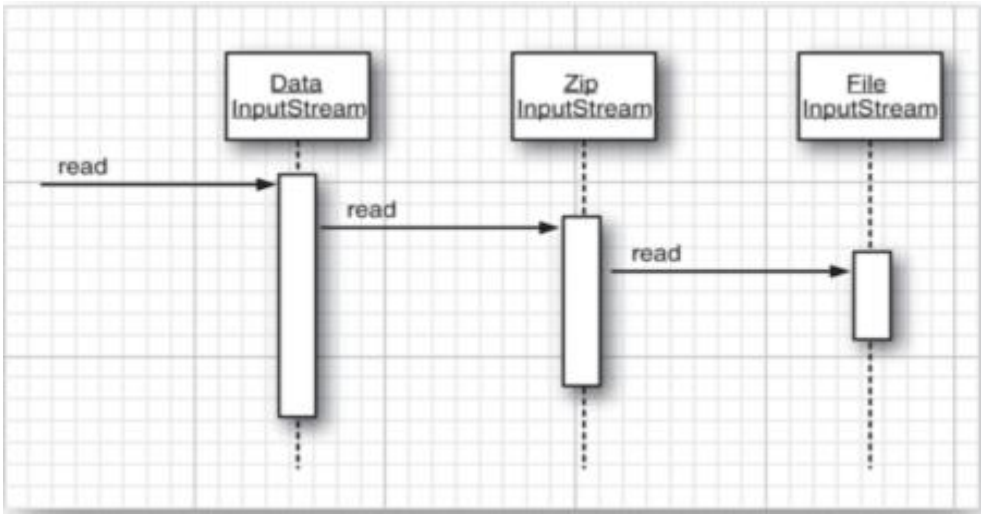


Figure 7. A sequence of filtered streams.

2.2 Demonstration of Stream and Files

java.io.InputStream 1.0

- abstract int read()

  reads a byte of data and returns the byte read. The read method returns a -1 at the end of the stream.

- int read(byte[] b)

  reads into an array of bytes and returns the actual number of bytes read, or -1 at the end

of the stream. The read method reads at most b.length bytes.

- int read(byte[] b, int off, int len)

  reads into an array of bytes. The read method returns the actual number of bytes read, or -1 at the end of the stream.

  Parameters: b The array into which the data is read

  off The offset into b where the first bytes should be placed

  len The maximum number of bytes to read

- long skip(long n)

  skips n bytes in the input stream. Returns the actual number of bytes skipped (which may be less than n if the end of the stream was encountered).

- int available()

  returns the number of bytes available without blocking. (Recall that blocking means that the current thread loses its turn.)

- void close()

  closes the input stream.

- void mark(int readlimit)

  puts a marker at the current position in the input stream. (Not all streams support this feature.) If more than readlimit bytes have been read from the input stream, then the stream is allowed to forget the marker.

- void reset()

  returns to the last marker. Subsequent calls to read reread the bytes. If there is no current marker, then the stream is not reset.

- boolean markSupported()

  returns true if the stream supports marking.

java.io.OutputStream 1.0

- abstract void write(int n)

writes a byte of data.

- void write(byte[] b)
- void write(byte[] b, int off, int len)

writes all bytes or a range of bytes in the array b.

Parameters: b The array from which to write the data

off The offset into b to the first byte that will be written

len The number of bytes to write

- void close()

flushes and closes the output stream.

- void flush()

flushes the output stream; that is, sends any buffered data to its destination.

java.io.Closeable 5.0

- void close()

closes this Closeable. This method may throw an IOException.

java.io.Flushable 5.0

- void flush()

flushes this Flushable.

java.lang.Readable 5.0

- int read(CharBuffer cb)

attempts to read as many char values into cb as it can hold. Returns the number of values read, or -1 if no further values are available from this Readable.

java.lang.Appendable 5.0

- Appendable append(char c)
- Appendable append(CharSequence cs)

appends the given code unit, or all code units in the given sequence, to this Appendable; returns this.

java.lang.CharSequence 1.4

- char charAt(int index)

returns the code unit at the given index.

- int length()

returns the number of code units in this sequence.

- CharSequence subSequence(int startIndex, int endIndex)

returns a CharSequence consisting of the code units stored at index startIndex to endIndex - 1.

- String toString()

returns a string consisting of the code units of this sequence.

java.io.FileInputStream 1.0

- FileInputStream(String name)
- FileInputStream(File file)

creates a new file input stream, using the file whose path name is specified by the name string or the file object. (The File class is described at the end of this chapter.) Path names that are not absolute are resolved relative to the working directory that was set when the VM started.

java.io.FileOutputStream 1.0

- FileOutputStream(String name)
- FileOutputStream(String name, boolean append)
- FileOutputStream(File file)
- FileOutputStream(File file, boolean append)

creates a new file output stream specified by the name string or the file object. (The File class is

described at the end of this chapter.) If the append parameter is true, then data are added at the end of the file. An existing file with the same name will not be deleted. Otherwise, this method deletes any existing file with the same name.

java.io.BufferedInputStream 1.0

- BufferedInputStream(InputStream in)

creates a buffered stream. A buffered input stream reads characters from a stream without causing a device access every time. When the buffer is empty, a new block of data is read into the buffer.

java.io.BufferedOutputStream 1.0

- BufferedOutputStream(OutputStream out)

creates a buffered stream. A buffered output stream collects characters to be written without causing a device access every time. When the buffer fills up or when the stream is flushed, the data are written.

java.io.PushbackInputStream 1.0

- PushbackInputStream(InputStream in)
- PushbackInputStream(InputStream in, int size)

constructs a stream with one-byte lookahead or a pushback buffer of specified size.

- void unread(int b)

pushes back a byte, which is retrieved again by the next call to read.

Parameters: b The byte to be read again

3 XML

3.1 Overview of XML

The Extensible Markup Language (XML) has replaced Java, Design Patterns, and Object Technology as the software industry's solution to world hunger." Indeed, as you will see in this chapter, XML is a very useful technology for describing structured information. XML tools make it easy to process and transform that information. However, XML is not a silver bullet. You need domain-specific standards and code libraries to use it effectively. Moreover, far from making Java technology obsolete, XML works very well with Java. Since the late 1990s, IBM, Apache, and others have been instrumental in producing high-quality Java libraries for XML processing. Starting with Java SE 1.4, Sun has integrated the most important libraries into the Java platform. A property file contains a set of name/value pairs, such as

```
fontname=Times Roman
fontsize=12
windowsize=400 200
color=0 50 100
```

You can use the *Properties* class to read in such a file with a single method call. That's a nice feature, but it doesn't really go far enough. In many cases, the information that you want to describe has more structure than the property file format can comfortably handle. Consider the `fontname/fontsize` entries in the example. It would be more object oriented to have a single entry:

```
font=Times Roman 12
```

But then parsing the font description gets ugly—you have to figure out when the font name ends and when the font size starts.

Property files have a single flat hierarchy. You can often see programmers work around that limitation with key names such as

```
title.fontname=Helvetica
title.fontsize=36
body.fontname=Times Roman
body.fontsize=12
```

Another shortcoming of the property file format is caused by the requirement that keys be unique. To store a sequence of values, you need another workaround, such as

```
menu.item.1=Times Roman
menu.item.2=Helvetica
menu.item.3=Goudy Old Style
```

The XML format solves these problems because it can express hierarchical structures and thus is more flexible than the flat table structure of a property file.

An XML file for describing a program configuration might look like this:

```
<configuration>
   <title>
      <font>
         <name>Helvetica</name>
         <size>36</size>
      </font>
   </title>
   <body>
      <font>
         <name>Times Roman</name>
         <size>12</size>
      </font>
   </body>
   <window>
      <width>400</width>
      <height>200</height>
   </window>
   <color>
      <red>0</red>
      <green>50</green>
      <blue>100</blue>
   </color>
   <menu>
      <item>Times Roman</item>
```

```
      <item>Helvetica</item>
      <item>Goudy Old Style</item>
   </menu>
</configuration>
```

SGML has been around since the 1970s for describing the structure of complex documents. It has been used with success in some industries that require ongoing maintenance of massive documentation, in particular, the aircraft industry. However, SGML is quite complex, so it has never caught on in a big way. Much of that complexity arises because SGML has two conflicting goals. SGML wants to make sure that documents are formed according to the rules for their document type, but it also wants to make data entry easy by allowing shortcuts that reduce typing. XML was designed as a simplified version of SGML for use on the Internet. As is often true, simpler is better, and XML has enjoyed the immediate and enthusiastic reception that has eluded SGML for so long.

Even though XML and HTML have common roots, there are important differences between the two.

- Unlike HTML, XML is case sensitive. For example, *<H1>* and *<h1>* are different XML tags.
- In HTML, you can omit end tags such as *</p>* or *
* tags if it is clear from the context where a paragraph or list item ends. In XML, you can never omit an end tag.
- In XML, elements that have a single tag without a matching end tag must end in a /, as in **. That way, the parser knows not to look for a *</img>* tag.
- In XML, attribute values must be enclosed in quotation marks. In HTML, quotation marks are optional. For example, *<applet code="MyApplet.class" width=300 height=300>* is legal HTML but not legal XML. In XML, you have to use quotation marks: *width="300"*.
- In HTML, you can have attribute names without values, such as *<input type="radio" name="language" value="Java" checked>*. In XML, all attributes must have values, such as *checked="true" or (ugh) checked="checked"*.

The Structure of an XML Document

An XML document should start with a header such as

```
<?xml version="1.0"?> or <?xml version="1.0" encoding="UTF-8"?>
```

Strictly speaking, a header is optional, but it is highly recommended. The header can be followed by a document type definition (DTD), such as

```
<!DOCTYPE web-app PUBLIC
   "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
   "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

DTDs are an important mechanism to ensure the correctness of a document, but they are not required. We discuss them later in this chapter.

Finally, the body of the XML document contains the root element, which can contain other elements. For example,

```
<?xml version="1.0"?>
<!DOCTYPE configuration . . .>
<configuration>
   <title>
      <font>
         <name>Helvetica</name>
         <size>36</size>
      </font>
   </title>
   . . .
</configuration>
```

An element can contain child elements, text, or both. In the preceding example, the `font` element has two child elements, *name* and *size*. The `name` element contains the text *"Helvetica"*. It is best if you structure your XML documents such that an element contains either child elements or text. In other words, you should avoid situations such as

```
<font>
   Helvetica
   <size>36</size>
</font>
```

This is called mixed contents in the XML specification. As you will see later in this chapter, you can simplify parsing if you avoid mixed contents.

XML elements can contain attributes, such as

```
<size unit="pt">36</size>
```

There is some disagreement among XML designers about when to use elements and when to use attributes. For example, it would seem easier to describe a font as

```
<font name="Helvetica" size="36"/>
```

than

```
<font>
   <name>Helvetica</name>
   <size>36</size>
</font>
```

However, attributes are much less flexible. Suppose you want to add units to the size value. If you use attributes, then you must add the unit to the attribute value:

```
<font name="Helvetica" size="36 pt"/>
```

Ugh! Now you have to parse the string `"36 pt"`, just the kind of hassle that XML was designed to avoid. Adding an attribute to the `size` element is much cleaner:

```
<font>
   <name>Helvetica</name>
   <size unit="pt">36</size>
</font>
```

A commonly used rule of thumb is that attributes should be used only to modify the interpretation of a value, not to specify values. If you find yourself engaged in metaphysical discussions about whether a particular setting is a modification of the interpretation of a value or not, then just say "no" to attributes and use elements throughout. Many useful XML documents don't use attributes at all.

In HTML, the rule for attribute usage is simple: If it isn't displayed on the web page, it's an attribute. For example, consider the hyperlink

```
<a href="http://java.sun.com">Java Technology</a>
```

The string *Java Technology* is displayed on the web page, but the URL of the link is not a part of the displayed page. However, the rule isn't all that helpful for most XML files because the data in an XML file aren't normally meant to be viewed by humans.

Elements and text are the "bread and butter" of XML documents. Here are a few other markup instructions that you might encounter:

- Character references have the form `&#`decimalValue`;` or `&#x`hexValue`;`. For example, the character é can be denoted with either of the following:

  ```
  &#233;
  &#xD9;
  ```

- Entity references have the form `&`name`;`. The entity references

  ```
  &lt;
  &gt;
  &amp;
  &quot;
  &apos;
  ```

  have predefined meanings: the less than, greater than, ampersand, quotation mark, and apostrophe characters. You can define other entity references in a DTD.

- CDATA sections are delimited by *<![CDATA[and]]>*. They are a special form of character data. You can use them to include strings that contain characters such as < > & without having them interpreted as markup, for example,

  ```
  <![CDATA[< & > are my favorite delimiters]]>
  ```

CDATA sections cannot contain the string *]]>*. Use this feature with caution! It is too often used as a back door for smuggling legacy data into XML documents.

- Processing instructions are instructions for applications that process XML documents. They are delimited by *<?* and *?>*, for example,

  ```
  <?xml-stylesheet href="mystyle.css" type="text/css"?>
  ```

  Every XML document starts with a processing instruction

  ```
  <?xml version="1.0"?>
  ```

- Comments are delimited by *<!--* and *-->*, for example,

  ```
  <!-- This is a comment. -->
  ```

  Comments should not contain the string --. Comments should only be information for human readers. They should never contain hidden commands. Use processing instructions for commands.

3.2 Demonstration of XML

An XML file for describing a program configuration might look like this:

```
<configuration>
   <title>
      <font>
         <name>Helvetica</name>
         <size>36</size>
      </font>
   </title>
   <body>
      <font>
         <name>Times Roman</name>
         <size>12</size>
      </font>
   </body>
   <window>
      <width>400</width>
      <height>200</height>
   </window>
```

```
   <color>
      <red>0</red>
      <green>50</green>
      <blue>100</blue>
   </color>
   <menu>
      <item>Times Roman</item>
      <item>Helvetica</item>
      <item>Goudy Old Style</item>
   </menu>
</configuration>
```

The XML format allows you to express the structure hierarchy and repeated elements without contortions.

As you can see, the format of an XML file is straightforward. It looks similar to an HTML file. There is a good reason—both the XML and HTML formats are descendants of the venerable Standard Generalized Markup Language (SGML).

4 Database Programming

This section discusses an overview of Database programming and the code demonstration.

4.1 Overview of Database Programming

The Design of JDBC

From the start, the developers of the Java technology at Sun were aware of the potential that Java showed for working with databases. In 1995, they began working on extending the standard Java library to deal with SQL access to databases. What they first hoped to do was to extend Java so that it could talk to any random database, using only "pure" Java. It didn't take them long to realize that this is an impossible task: There are simply too many databases out there, using too many protocols. Moreover, although database vendors were all in favor of Sun providing a standard network protocol for database access, they were only in favor of it if Sun decided to use their network protocol.

What all the database vendors and tool vendors did agree on was that it would be useful if Sun provided a pure Java API for SQL access along with a driver manager to allow third-party

drivers to connect to specific databases. Database vendors could provide their own drivers to plug in to the driver manager. There would then be a simple mechanism for registering third-party drivers with the driver manager. As a result, two APIs were created. Application programmers use the JDBC API, and database vendors and tool providers use the JDBC Driver API.

This organization follows the very successful model of Microsoft's ODBC, which provided a C programming language interface for database access. Both JDBC and ODBC are based on the same idea: Programs written according to the API talk to the driver manager, which, in turn, uses a driver to talk to the actual database.

All this means the JDBC API is all that most programmers will ever have to deal with—see Figure 8.

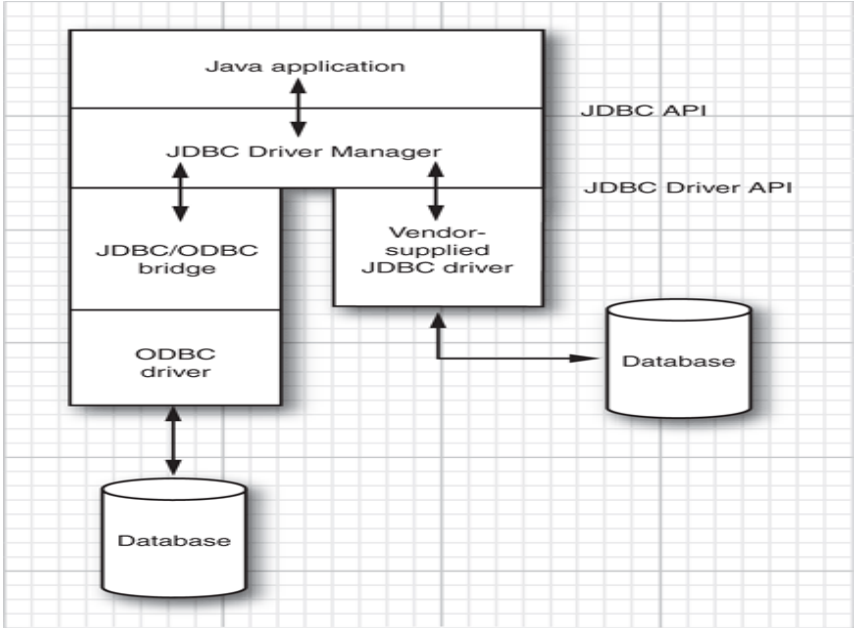


Figure 8. JDBC-to-database communication path

JDBC Driver Types

The JDBC specification classifies drivers into the following types:

- A type 1 driver translates JDBC to ODBC and relies on an ODBC driver to communicate with the database. Sun included one such driver, the JDBC/ODBC bridge, with earlier versions of the JDK. However, the bridge requires deployment and proper configuration of an ODBC driver. When JDBC was first released, the bridge was handy for testing, but it was never intended for production use. At this point, many better drivers are available, and we advise against using the JDBC/ODBC bridge.

- A type 2 driver is written partly in Java and partly in native code; it communicates with the client API of a database. When you use such a driver, you must install some platform-specific code onto the client in addition to a Java library.

- A type 3 driver is a pure Java client library that uses a database-independent protocol to communicate database requests to a server component, which then translates the requests into a database-specific protocol. This can simplify deployment because the platform-specific code is located only on the server.

- A type 4 driver is a pure Java library that translates JDBC requests directly to a database-specific protocol.

Most database vendors supply either a type 3 or type 4 driver with their database. Furthermore, a number of third-party companies specialize in producing drivers with better standards conformance, support for more platforms, better performance, or, in some cases, simply better reliability than the drivers that are provided by the database vendors.

In summary, the ultimate goal of JDBC is to make possible the following:

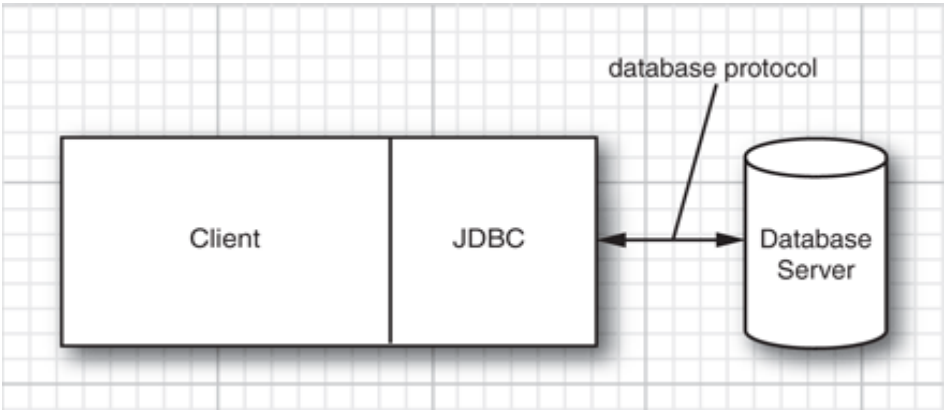
- Programmers can write applications in the Java programming language to access any database, using standard SQL statements—or even specialized extensions of SQL—while still following Java language conventions.

- Database vendors and database tool vendors can supply the low-level drivers. Thus, they can optimize their drivers for their specific products.

Typical Uses of JDBC

The traditional client/server model has a rich GUI on the client and a database on the server (see Figure 9). In this model, a JDBC driver is deployed on the client.

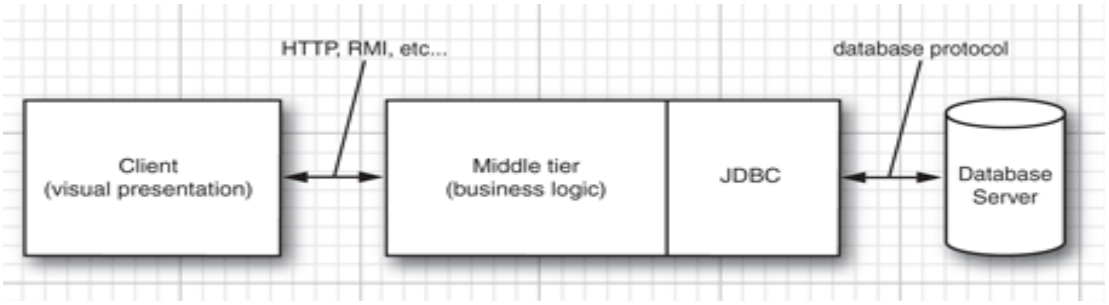
Figure 9. A traditional client/server application.



However, the world is moving away from client/server and toward a three-tier model or even more advanced n-tier models. In the three-tier model, the client does not make database calls. Instead, it calls on a middleware layer on the server that in turn makes the database queries. The three-tier model has a couple of advantages. It separates visual presentation (on the client) from the business logic (in the middle tier) and the raw data (in the database). Therefore, it becomes possible to access the same data and the same business rules from multiple clients, such as a Java application or applet or a web form.

Communication between the client and middle tier can occur through HTTP (when you use a web browser as the client) or another mechanism such as remote method invocation (RMI. JDBC manages the communication between the middle tier and the back-end database. Figure 10 shows the basic architecture. There are, of course, many variations of this model. In particular, the Java Enterprise Edition defines a structure for application servers that manage code modules called Enterprise JavaBeans, and provides valuable services such as load balancing, request caching, security, and object-relational mapping. In that architecture, JDBC still plays an important role for issuing complex database queries. (For more information on the Enterprise Edition, see http://java.sun.com/javaee.)

Figure 10. A three-tier application.



4.2 Demonstration of Database Programming

```
//SQLSERVER
import java.sql.*;
import java.net.*;
import java.io.*;

public class SqlServer
{
  public static void main(String []args)
   {
          BufferedReader br;
         ServerSocket ss;
         Socket s;
         int portno;
         String qry;
         Connection con;
      try
        {
          br=new BufferedReader(new InputStreamReader(System.in));
          System.out.println("enter portno:");
          portno=Integer.parseInt(br.readLine());

          ss=new ServerSocket(portno);
          System.out.println("Server waiting...");

          Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

     while(true)
       {
          s=ss.accept();
          new SqlServerReadData(s);
```

```
        }
    } catch(Exception e){}
 }
}


class SqlServerReadData implements Runnable
{
  Connection con;
Statement stmt;
ResultSet r;
ResultSetMetaData rm;

Socket s;
BufferedReader is;
PrintWriter os;
String qry,result,uname,pass,dsn;
Thread t;
int colcount;

SqlServerReadData(Socket s)
{
try
 {
 this.s=s;
is=new BufferedReader(new InputStreamReader(s.getInputStream()));
os=new PrintWriter(s.getOutputStream());
//System.out.println("Username");
//uname=is.readLine();
//System.out.println("Passward");
//pass=is.readLine();
//System.out.println("dsn");
dsn=is.readLine();

String str="jdbc:odbc:"+dsn;
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
con=DriverManager.getConnection(str,"","");
stmt=con.createStatement();
t=new Thread(this);
t.start();
}
catch(Exception e){}
}

public void run()
{
```

```
try
{
while(true)
{
try
{
qry=is.readLine();
if(qry.compareToIgnoreCase("quit")==0)
break;
if(!(qry.startsWith("select")))
{
 try
{
r=stmt.executeQuery(qry);
os.flush();
}catch(Exception e){}
}
else
try
{
r=stmt.executeQuery(qry);
rm=r.getMetaData();
colcount=rm.getColumnCount();

for(int i=1;i<=colcount;i++)
{
result=rm.getColumnLabel(i);
os.print(result);
os.flush();
}
os.println("");
for(int i=1;i<=34;i++)
os.print("-");
os.println("");
while(r.next())
{
result ="";
for(int i=1;i<=colcount;i++)
{
result=result+"       "+r.getString(i);
}

os.println(result);
os.flush();
}
r.close();
```

```
}
catch(SQLException e){}
catch(Exception e){}
os.println("end");
os.flush();
}
catch(Exception e){}
}
stmt.close();
}
catch(Exception e){}
}
}


//SQLCLIENT
import java.sql.*;
import java.net.*;
import java.io.*;

public class SqlClient
{
  public static void main(String []args)
   {
    try
      {
          BufferedReader br=new BufferedReader(new
InputStreamReader(System.in));;
          BufferedReader is;
          PrintWriter os;
         Socket s;
         int portno;
         String qry,hostname,result,uname,pass,dsn;

  System.out.print("enter the hostname:");
    hostname=br.readLine();

 System.out.println("enter portno:");
 portno=Integer.parseInt(br.readLine());

s=new Socket(hostname,portno);
 is =new BufferedReader(new InputStreamReader(s.getInputStream()));
os=new PrintWriter(s.getOutputStream());
```

```
System.out.println("Username");
uname=is.readLine();
System.out.println("Passward");
pass=is.readLine();
System.out.println("dsn");
dsn=br.readLine();

os.println(uname);
os.flush();

os.println(pass);
os.flush();

os.println(dsn);
os.flush();

while(true)
{
 System.out.print("SQL:\>");
 qry=br.readLine();
if(qry.equals("quit"))
break;

os.println(qry);
os.flush();

result=is.readLine();
while(!(result.equals("end")))
{
 System.out.println(result);
 result=is.readLine();
}
}

os.println("quit");
os.flush();
os.close();
is.close();
s.close();
}
catch(Exception e){}
}
```

5 Internationalization

This section discusses an overview of Internationalization and the code demonstration.

5.1 Overview of Internationalization

The Java programming language was the first language designed from the ground up to support internationalization. From the beginning, it had the one essential feature needed for effective internationalization: It used Unicode for all strings. Unicode support makes it easy to write programs in the Java programming language that manipulate strings in any one of multiple languages.

Many programmers believe that all they need to do to internationalize their application is to support Unicode and to translate the messages in the user interface. However, as this chapter demonstrates, there is a lot more to internationalizing programs than just Unicode support. Dates, times, currencies—even numbers—are formatted differently in different parts of the world. You need an easy way to configure menu and button names, message strings, and keyboard shortcuts for different languages.

In this chapter, we show you how to write internationalized Java applications and applets and how to localize date, time, numbers, text, and GUIs. We show you tools that Java offers for writing internationalized programs. We close this chapter with a complete example, a retirement calculator applet that can change how it displays its results depending on the location of the machine that is downloading it.

Locales

The Java platform does not require you to use the same Locale throughout your program. If you wish, you can assign a different Locale to every locale-sensitive object in your program. This flexibility allows you to develop multilingual applications, which can display information in multiple languages.

However, most applications are not multi-lingual and their locale-sensitive objects rely on the default Locale. Set by the Java Virtual Machine when it starts up, the default Locale corresponds to the locale of the host platform. To determine the default Locale of your Java Virtual Machine, invoke the Locale.getDefault method. You should not set the default Locale programmatically because it is shared by all locale-sensitive classes.

Distributed computing raises some interesting issues. For example, suppose you are designing an application server that will receive requests from clients in various countries. If the Locale for each client is different, what should be the Locale of the server? Perhaps the server is multithreaded, with each thread set to the Locale of the client it services. Or perhaps all data passed between the server and the clients should be locale-independent.

Which design approach should you take? If possible, the data passed between the server and the clients should be locale-independent. This simplifies the design of the server by making the clients responsible for displaying the data in a locale-sensitive manner. However, this approach won't work if the server must store the data in a locale-specific form. For example, the server might store Spanish, English, and French versions of the same data in different database columns. In this case, the server might want to query the client for its Locale, since the Locale may have changed since the last request.

When you look at an application that is adapted to an international market, the most obvious difference you notice is the language. This observation is actually a bit too limiting for true internationalization: Countries can share a common language, but you still might need to do some work to make computer users of both countries happy.

In all cases, menus, button labels, and program messages will need to be translated to the local language; they might also need to be rendered in a different script. There are many more subtle differences; for example, numbers are formatted quite differently in English and in German. The number

123,456.78

should be displayed as

123.456,78

for a German user. That is, the role of the decimal point and the decimal comma separator are reversed. There are similar variations in the display of dates. In the United States, dates are somewhat irrationally displayed as month/day/year. Germany uses the more sensible order of day/month/year, whereas in China, the usage is year/month/day. Thus, the date

3/22/61

should be presented as

22.03.1961

to a German user. Of course, if the month names are written out explicitly, then the difference in languages becomes apparent. The English

March 22, 1961

should be presented as

22. März 1961

in German, or

1961年3月22日

in Chinese.

There are several formatter classes that take these differences into account. To control the formatting, you use the Locale class. A locale describes

- A language.
- Optionally, a location.
- Optionally, a variant.

For example, in the United States, you use a locale with

language=English, location=United States.

In Germany, you use a locale with

language=German, location=Germany.

Switzerland has four official languages (German, French, Italian, and Rhaeto-Romance). A German speaker in Switzerland would want to use a locale with

language=German, location=Switzerland

This locale would make formatting work similarly to how it would work for the German locale; however, currency values would be expressed in Swiss francs, not German marks.

If you only specify the language, say,

language=German

then the locale cannot be used for country-specific issues such as currencies.

Variants are, fortunately, rare and are needed only for exceptional or system-dependent situations. For example, the Norwegians are having a hard time agreeing on the spelling of their language (a derivative of Danish). They use two spelling rule sets: a traditional one called Bokmål and a new one called Nynorsk. The traditional spelling would be expressed as a variant

language=Norwegian, location=Norway, variant=Bokmål

To express the language and location in a concise and standardized manner, the Java programming language uses codes that were defined by the International Organization for Standardization (ISO). The local language is expressed as a lowercase two-letter code, following ISO 639-1, and the country code is expressed as an uppercase two-letter code, following ISO 3166-1. Tables 1 and 2 show some of the most common codes.

Table 1. Common ISO 639-1 Language Codes.

Language	Code
Chinese	zh
Danish	da
Dutch	nl
English	en
French	fr
Finnish	fi
German	de
Greek	el
Italian	it
Japanese	ja
Korean	ko
Norwegian	no
Portuguese	pt
Spanish	sp
Swedish	sv
Turkish	tr

Table 2. Common ISO 3166-1 Country Codes.

Country	Code
Austria	AT
Belgium	BE
Canada	CA
China	CN
Denmark	DK
Finland	FI
Germany	DE
Great Britain	GB
Greece	GR
Ireland	IE
Italy	IT
Japan	JP
Korea	KR
The Netherlands	NL
Norway	NO
Portugal	PT
Spain	ES
Sweden	SE
Switzerland	CH
Taiwan	TW
Turkey	TR
United States	US

These codes do seem a bit random, especially because some of them are derived from local languages (German = Deutsch = *de*, Chinese = zhongwen = *zh*), but at least they are standardized.

To describe a locale, you concatenate the language, country code, and variant (if any) and pass this string to the constructor of the *Locale* class.

```
Locale german = new Locale("de");
Locale germanGermany = new Locale("de", "DE");
Locale germanSwitzerland = new Locale("de", "CH");
Locale norwegianNorwayBokmål = new Locale("no", "NO", "B");
```

For your convenience, Java SE predefines a number of locale objects:

```
Locale.CANADA
Locale.CANADA_FRENCH
Locale.CHINA
Locale.FRANCE
Locale.GERMANY
Locale.ITALY
Locale.JAPAN
Locale.KOREA
Locale.PRC
Locale.TAIWAN
Locale.UK
Locale.US
```

Java SE also predefines a number of language locales that specify just a language without a location:

```
Locale.CHINESE
Locale.ENGLISH
Locale.FRENCH
Locale.GERMAN
Locale.ITALIAN
Locale.JAPANESE
Locale.KOREAN
Locale.SIMPLIFIED_CHINESE
Locale.TRADITIONAL_CHINESE
```

Besides constructing a locale or using a predefined one, you have two other methods for obtaining a locale object.

The static `getDefault` method of the `Locale` class initially gets the default locale as stored by the local operating system. You can change the default Java locale by calling `setDefault`; however, that change only affects your program, not the operating system. Similarly, in an applet, the `getLocale` method returns the locale of the user viewing the applet. Finally, all locale-dependent utility classes can return an array of the locales they support. For example,

```
Locale[] supportedLocales = DateFormat.getAvailableLocales();
```

returns all locales that the *DateFormat* class can handle.

Once you have a locale, what can you do with it? Not much, as it turns out. The only useful methods in the Locale class are the ones for identifying the language and country codes. The most important one is getDisplayName. It returns a string describing the locale. This string does not contain the cryptic two-letter codes, but it is in a form that can be presented to a user, such as

```
German (Switzerland)
```

Actually, there is a problem here. The display name is issued in the default locale. That might not be appropriate. If your user already selected German as the preferred language, you probably want to present the string in German. You can do just that by giving the German locale as a parameter: The code

```
Locale loc = new Locale("de", "CH");
System.out.println(loc.getDisplayName(Locale.GERMAN));
```

prints

```
Deutsch (Schweiz)
```

This example shows why you need Locale objects. You feed it to locale-aware methods that produce text that is presented to users in different locations. You can see many examples in the following sections.

Locale-Sensitive Services SPI

This feature enables the plug-in of locale-dependent data and services. In this way, third parties are able to provide implementations of most locale-sensitive classes in the *java.text* and *java.util* packages.

The implementation of SPIs *(Service Provider Interface)* is based on abstract classes and Java interfaces that are implemented by the service provider. At runtime the Java class loading mechanism is used to dynamically locate and load classes that implement the SPI.

You can use the locale-sensitive services SPI to provide the following locale sensitive implementations:

- *BreakIterator* objects

- Collator objects
- Language code, Country code, and Variant name for the Locale class
- Time Zone names
- Currency symbols
- *DateFormat* objects
- *DateFormatSymbol* objects
- *NumberFormat* objects
- *DecimalFormatSymbols* objects

The corresponding SPIs are contained both in *java.text.spi* and in *java.util.spi* packages:

java.util.spi	**java.text.spi**
• *CurrencyNameProvider* • *LocaleServiceProvider* • *TimeZoneNameProvider*	• *BreakIteratorProvider* • *CollatorProvider* • *DateFormatProvider* • *DateFormatSymbolsProvider* • *DecimalFormatSymbolsProvider* • *NumberFormatProvider*

For example, if you would like to provide a *NumberFormat* object for a new locale, you have to implement the *java.text.spi.NumberFormatProvider class*. You need to extend this class and implement its methods:

- *getCurrencyInstance(Locale locale)*
- *getIntegerInstance(Locale locale)*
- *getNumberInstance(Locale locale)*
- *getPercentInstance(Locale locale)*

```
Locale loc = new Locale("da", "DK");

NumberFormat nf = NumberFormatProvider.getNumberInstance(loc);
```

These methods first check whether the Java runtime environment supports the requested locale; if so, they use that support. Otherwise, the methods call the getAvailableLocales() methods of installed providers for the appropriate interface to find a provider that supports the requested locale.

5.2 Demonstration of Internationalization

java.util.Locale 1.1

- Locale(String language)
- Locale(String language, String country)
- Locale(String language, String country, String variant)

constructs a locale with the given language, country, and variant.

- static Locale getDefault()

returns the default locale.

- static void setDefault(Locale loc)

sets the default locale.

- String getDisplayName()

returns a name describing the locale, expressed in the current locale.

- String getDisplayName(Locale loc)

returns a name describing the locale, expressed in the given locale.

- String getLanguage()

returns the language code, a lowercase two-letter ISO-639 code.

- String getDisplayLanguage()

returns the name of the language, expressed in the current locale.

- String getDisplayLanguage(Locale loc)

returns the name of the language, expressed in the given locale.

- String getCountry()

returns the country code as an uppercase two-letter ISO-3166 code.

- String getDisplayCountry()

returns the name of the country, expressed in the current locale.

- String getDisplayCountry(Locale loc)

returns the name of the country, expressed in the given locale.

- String getVariant()

returns the variant string.

- String getDisplayVariant()

returns the name of the variant, expressed in the current locale.

- String getDisplayVariant(Locale loc)

returns the name of the variant, expressed in the given locale.

- String toString()

returns a description of the locale, with the language, country, and variant separated by underscores (e.g., "de_CH").

java.awt.Applet 1.0

- Locale getLocale() [1.1]

gets the locale for this applet.

6 Advanced Swing

Java Swing is a GUI toolkit for Java. Swing is one part of the Java Foundation Classes (JFC). Swing includes graphical user interface (GUI) widgets such as text boxes, buttons, split-panes, and tables. Swing widgets provide more sophisticated GUI components than the earlier Abstract

Window Toolkit. Since they are written in pure Java, they run the same on all platforms, unlike the AWT which is tied to the underlying platform's windowing system. Swing supports pluggable look and feel – not by using the native platform's facilities, but by roughly emulating them. This means you can get any supported look and feel on any platform. The disadvantage of lightweight components is possibly slower execution. The advantage is uniform behavior on all platforms.

History

The Internet Foundation Classes (IFC) were a graphics library for Java originally developed by Netscape Communications Corporation and first released on Dec 16, 1996.

On April 2, 1996, Sun Microsystems and Netscape Communications Corporation announced their intention to combine IFC with other technologies to form the Java Foundation Classes. In addition to the components originally provided by IFC, Swing introduced a mechanism that allowed the look and feel of every component in an application to be altered without making substantial changes to the application code. The introduction of support for a pluggable look and feel allowed Swing components to emulate the appearance of native components while still retaining the benefits of platform independence. This feature also makes it easy to have an individual application's appearance look very different from other native programs.

Originally distributed as a separately downloadable library, Swing has been included as part of the Java Standard Edition since release 1.2. The Swing classes are contained in the *javax.swing* package hierarchy.

Architecture

The Swing library makes heavy use of the Model/View/Controller software design pattern, which attempts to separate the data being viewed from the method by which it is viewed. Because of this, most Swing components have associated models (typically as interfaces), and the programmer can use various default implementations or provide their own. For example, the JTable has a model called TableModel that describes an interface for how a table would access tabular data. A default implementation of this operates on a two-dimensional array.

Swing favors relative layouts (which specify the positional relationships between components), as opposed to absolute layouts (which specify the exact location and size of components). The

motivation for this is to allow Swing applications to work and appear visually correct regardless of the underlying systems colors, fonts, language, sizes or I/O devices. This can make screen design somewhat difficult and numerous tools have been developed to allow visual designing of screens.

Swing also uses a publish subscribe event model (as does AWT), where listeners subscribe to events that are fired by the application (such as pressing a button, entering text or clicking a checkbox). The model classes typically include, as part of their interface, methods for attaching listeners (this is the publish aspect of the event model). The frequent use of loose coupling within the framework makes Swing programming somewhat different from higher-level GUI design languages and 4GLs. This is a contributing factor to Swing having such a steep learning curve.

Look and feel

Swing allows one to specialize the look and feel of widgets, by modifying the default (via runtime parameters), deriving from an existing one, by creating one from scratch, or, beginning with J2SE 5.0, by using the skinnable Synth Look and Feel, which is configured with an XML property file. The look and feel can be changed at runtime, and early demonstrations of Swing would frequently provide a way to do this.

Relationship to AWT

Since early versions of Java, a portion of the Abstract Windowing Toolkit (AWT) has provided platform independent APIs for user interface components. In AWT, each component is rendered and controlled by a native peer component specific to the underlying windowing system.
By contrast, Swing components are often described as lightweight because they do not require allocation of native resources in the operating system's windowing toolkit. The AWT components are referred to as heavyweight components. Much of the Swing API is generally a complementary extension of the AWT rather than a direct replacement. In fact, every Swing lightweight interface ultimately exists within an AWT heavyweight component because all of the top-level components in Swing (JApplet, JDialog, JFrame, and JWindow) extend an AWT top-level container. The core rendering functionality used by Swing to draw its lightweight components is provided by Java2D, another part of JFC. However, the use of both lightweight

and heavyweight components within the same window is generally discouraged due to Z-order incompatibilities.

Relationship to SWT

The Standard Widget Toolkit (SWT) is a competing toolkit originally developed by IBM and now maintained by the Eclipse Foundation. SWT's implementation has more in common with the heavyweight components of AWT. This confers benefits such as more accurate fidelity with the underlying native windowing toolkit, at the cost of an increased exposure to the native resources in the programming model.

The advent of SWT has given rise to a great deal of division among Java desktop developers with many strongly favouring either SWT or Swing. A renewed focus on Swing look and feel fidelity with the native windowing toolkit in the approaching Java SE 6 release (as of February 2006) is probably a direct result of this.

Example

The following is a Hello World program using Swing.

```
import javax.swing.JFrame;
import javax.swing.JLabel;
public final class HelloWorld extends JFrame {
private HelloWorld() {
setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
getContentPane().add(new JLabel("Hello, World!"));
pack();
setLocationRelativeTo(null);
}
public static void main(String[] args) {
new HelloWorld().setVisible(true);
}
}
```

6.1 Overview of Advanced Swing

Lists

If you want to present a set of choices to a user, and a radio button or checkbox set consumes too much space, you can use a combo box or a list. Combo boxes were covered in Volume I because they are relatively simple. The JList component has many more features, and its design is similar to that of the tree and table components. For that reason, it is our starting point for the discussion of complex Swing components.

Of course, you can have lists of strings, but you can also have lists of arbitrary objects, with full control of how they appear. The internal architecture of the list component that makes this generality possible is rather elegant. Unfortunately, the designers at Sun felt that they needed to show off that elegance, rather than hiding it from the programmer who just wants to use the component. You will find that the list control is somewhat awkward to use for common cases because you need to manipulate some of the machinery that makes the general cases possible. We walk you through the simple and most common case, a list box of strings, and then give a more complex example that shows off the flexibility of the list component.

The JList Component

The JList component shows a number of items inside a single box. Figure 11 shows an admittedly silly example. The user can select the attributes for the fox, such as "quick," "brown," "hungry," "wild," and, because we ran out of attributes, "static," "private," and "final." You can thus have the static final fox jump over the lazy dog.

Figure 11. A List Box

To construct this list component, you first start out with an array of strings, then pass the array to the JList constructor:

```
String[] words= { "quick", "brown", "hungry", "wild", ... };

JList wordList = new JList(words);
```

Alternatively, you can use an anonymous array:

```
JList wordList = new JList(new String[] {"quick", "brown", "hungry",
"wild", ... });
```

List boxes do not scroll automatically. To make a list box scroll, you must insert it into a scroll pane:

```
JScrollPane scrollPane = new JScrollPane(wordList);
```

You then add the scroll pane, not the list, into the surrounding panel. We must admit that the separation of the list display and the scrolling mechanism is elegant in theory, but it is a pain in practice. Essentially all lists that we ever encountered needed scrolling. It seems cruel to force programmers to go through hoops in the default case just so they can appreciate that elegance.

By default, the list component displays eight items; use the *setVisibleRowCount* method to change that value:

```
wordList.setVisibleRowCount(4); // display 4 items
```

You can set the layout orientation to one of three values:

- JList.VERTICAL (the default)— Arrange all items vertically.
- JList.VERTICAL_WRAP— Start new columns if there are more items than the visible row count (see Figure 12).

Figure 12. Lists with vertical and horizontal wrap

- JList.HORIZONTAL_WRAP— Start new columns if there are more items than the visible row count, but fill them horizontally. Look at the placement of the words "quick," "brown," and "hungry" in Figure 12 to see the difference between vertical and horizontal wrap.

By default, a user can select multiple items. To add more items to a selection, press the CTRL key while clicking on each item. To select a contiguous range of items, click on the first one, then hold down the SHIFT key and click on the last one.

You can also restrict the user to a more limited selection mode with the *setSelectionMode* method:

```
wordList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION); // select one item at a time
```

```
wordList.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION
); // select one item or one range of items
```

You might recall from Volume I that the basic user interface components send out action events when the user activates them. List boxes use a different notification mechanism. Rather than listening to action events, you need to listen to list selection events. Add a list selection listener to the list component, and implement the method

```
public void valueChanged(ListSelectionEvent evt)
```

in the listener.

When the user selects items, a flurry of list selection events is generated. For example, suppose the user clicks on a new item. When the mouse button goes down, an event reports a change in selection. This is a transitional event—the call

```
event.isAdjusting()
```

returns true if the selection is not yet final. Then, when the mouse button goes up, there is another event, this time with isAdjusting returning false. If you are not interested in the transitional events, then you can wait for the event for which isAdjusting is false. However, if you want to give the user instant feedback as soon as the mouse button is clicked, you need to process all events.

Once you are notified that an event has happened, you will want to find out what items are currently selected. The getSelectedValues method returns an array of objects containing all selected items. Cast each array element to a string.

```
Object[] values = list.getSelectedValues();

for (Object value : values)

   do something with (String) value;
```

You cannot cast the return value of getSelectedValues from an Object[] array to a String[] array. The return value was not created as an array of strings, but as an array of objects, each of which happens to be a string. To process the return value as an array of strings, use the following code:

```
int length = values.length;

String[] words = new String[length];

System.arrayCopy(values, 0, words, 0, length);
```

If your list does not allow multiple selections, you can call the convenience method *getSelectedValue*. It returns the first selected value (which you know to be the only value if multiple selections are disallowed).

```
String value = (String) list.getSelectedValue();
```

List components do not react to double clicks from a mouse. As envisioned by the designers of Swing, you use a list to select an item, and then you click a button to make something happen. However, some user interfaces allow a user to double-click on a list item as a shortcut for item selection and acceptance of a default action. If you want to implement this behavior, you have to add a mouse listener to the list box, then trap the mouse event as follows:

```
public void mouseClicked(MouseEvent evt){

   if (evt.getClickCount() == 2)

   {

      JList source = (JList) evt.getSource();

      Object[] selection = source.getSelectedValues();

      doAction(selection);

   }

}
```

List Models

In the preceding section, you saw the most common method for using a list component:

1. Specify a fixed set of strings for display in the list.

2. Place the list inside a scroll pane.
3. Trap the list selection events.

In the remainder of the section on lists, we cover more complex situations that require a bit more finesse:

- Very long lists
- Lists with changing contents
- Lists that don't contain strings

In the first example, we constructed a *JList* component that held a fixed collection of strings. However, the collection of choices in a list box is not always fixed. How do we add or remove items in the list box? Somewhat surprisingly, there are no methods in the *JList* class to achieve this. Instead, you have to understand a little more about the internal design of the list component. The list component uses the model-view-controller design pattern to separate the visual appearance (a column of items that are rendered in some way) from the underlying data (a collection of objects).

The *JList* class is responsible for the visual appearance of the data. It actually knows very little about how the data are stored—all it knows is that it can retrieve the data through some object that implements the *ListModel* interface:

```
public interface ListModel
{
   int getSize();
   Object getElementAt(int i);
   void addListDataListener(ListDataListener l);
   void removeListDataListener(ListDataListener l);
}
```

Through this interface, the *JList* can get a count of elements and retrieve each one of the elements. Also, the *JList* object can add itself as a *ListDataListener*. That way, if the collection of elements changes, the *JList* gets notified so that it can repaint itself.

Why is this generality useful? Why doesn't the *JList* object simply store an array of objects?

Note that the interface doesn't specify how the objects are stored. In particular, it doesn't force them to be stored at all! The *getElementAt* method is free to recompute each value whenever it is called. This is potentially useful if you want to show a very large collection without having to store the values.

Here is a somewhat silly example: We let the user choose among all three-letter words in a list box (see Figure 13).

Figure 6-3. Choosing from a very long list of selections

There are 26 x 26 x 26 = 17,576 three-letter combinations. Rather than storing all these combinations, we recompute them as requested when the user scrolls through them.

This turns out to be easy to implement. The tedious part, adding and removing listeners, has been done for us in the *AbstractListModel* class, which we extend. We only need to supply the *getSize* and *getElementAt* methods:

```
class WordListModel extends AbstractListModel
{
   public WordListModel(int n) { length = n; }
   public int getSize() { return (int) Math.pow(26, length); }
   public Object getElementAt(int n)
   {
```

```
      // compute nth string
      . . .
   }
   . . .
}
```

The computation of the nth string is a bit technical—you'll find the details in *the long list test* in the code demonstration section below.

Now that we have supplied a model, we can simply build a list that lets the user scroll through the elements supplied by the model:

```
JList wordList = new JList(new WordListModel(3));
wordList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
JScrollPane scrollPane = new JScrollPane(wordList);
```

The point is that the strings are never stored. Only those strings that the user actually requests to see are generated.

We must make one other setting. We must tell the list component that all items have a fixed width and height. The easiest way to set the cell dimensions is to specify a prototype cell value:

```
wordList.setPrototypeCellValue("www");
```

The prototype cell value is used to determine the size for all cells. (We use the string "www" because "w" is the widest lowercase letter in most fonts.)

Alternatively, you can set a fixed cell size:

```
wordList.setFixedCellWidth(50);
wordList.setFixedCellHeight(15);
```

If you don't set a prototype value or a fixed cell size, the list component computes the width and height of each item. That can take a long time.

As a practical matter, very long lists are rarely useful. It is extremely cumbersome for a user to scroll through a huge selection. For that reason, we believe that the list control has been completely overengineered. A selection that a user can comfortably manage on the screen is certainly small enough to be stored directly in the list component. That arrangement would have saved programmers from the pain of having to deal with the list model as a separate entity. On the other hand, the *JList* class is consistent with the *JTree* and *JTable* class where this generality is useful.

Inserting and Removing Values

You cannot directly edit the collection of list values. Instead, you must access the model and then add or remove elements. That, too, is easier said than done. Suppose you want to add more values to a list. You can obtain a reference to the model:

```
ListModel model = list.getModel();
```

But that does you no good—as you saw in the preceding section, the *ListModel* interface has no methods to insert or remove elements because, after all, the whole point of having a list model is that it need not store the elements.

Let's try it the other way around. One of the constructors of JList takes a vector of objects:

```
Vector<String> values = new Vector<String>();
values.addElement("quick");
values.addElement("brown");
. . .
JList list = new JList(values);
```

You can now edit the vector and add or remove elements, but the list does not know that this is happening, so it cannot react to the changes. In particular, the list cannot update its view when you add the values. Therefore, this constructor is not very useful.

Instead, you should construct a *DefaultListModel* object, fill it with the initial values, and associate it with the list. The *DefaultListModel* class implements the *ListModel* interface and manages a collection of objects.

```
DefaultListModel model = new DefaultListModel();

model.addElement("quick");

model.addElement("brown");

. . .

JList list = new JList(model);
```

Now you can add or remove values from the model object. The model object then notifies the list of the changes, and the list repaints itself.

```
model.removeElement("quick");

model.addElement("slow");
```

For historical reasons, the *DefaultListMode*l class doesn't use the same method names as the collection classes.

The default list model uses a vector internally to store the values.

There are JList constructors that construct a list from an array or vector of objects or strings. You might think that these constructors use a DefaultListModel to store these values. That is not the case—the constructors build a trivial model that can access the values without any provisions for notification if the content changes. For example, here is the code for the constructor that constructs a JList from a Vector:

```
public JList(final Vector<?> listData)

{

   this (new AbstractListModel()

   {

      public int getSize() { return listData.size(); }

      public Object getElementAt(int i) { return
listData.elementAt(i); }

   });

}
```

That means, if you change the contents of the vector after the list is constructed, then the list might show a confusing mix of old and new values until it is completely repainted. (The keyword final in the preceding constructor does not prevent you from changing the vector elsewhere—it only means that the constructor itself won't modify the value of the *listData* reference; the keyword is required because the *listData* object is used in the inner class.)

javax.swing.JList 1.2

- ListModel getModel()

gets the model of this list.

javax.swing.DefaultListModel 1.2

- void addElement(Object obj)

adds the object to the end of the model.

- boolean removeElement(Object obj)

removes the first occurrence of the object from the model. Returns true if the object was contained in the model, false otherwise.

Rendering Values

So far, all lists that you have seen in this chapter contained strings. It is actually just as easy to show a list of icons—simply pass an array or vector filled with Icon objects. More interestingly, you can easily represent your list values with any drawing whatsoever.

Although *the JList* class can display strings and icons automatically, you need to install a list cell renderer into *the JList* object for all custom drawing. A list cell renderer is any class that implements the following interface:

```
interface ListCellRenderer

{

   Component getListCellRendererComponent(JList list, Object value,
int index,

      boolean isSelected, boolean cellHasFocus);

}
```

This method is called for each cell. It returns a component that paints the cell contents. The component is placed at the appropriate location whenever a cell needs to be rendered.

One way to implement a cell renderer is to create a class that extends JComponent, like this:

```
class MyCellRenderer extends JComponent implements ListCellRenderer

{

   public Component getListCellRendererComponent(JList list, Object
value, int index,

      boolean isSelected, boolean cellHasFocus)

   {

      // stash away information that is needed for painting and size
measurement

      return this;

   }

   public void paintComponent(Graphics g)

   {

      // paint code goes here

   }

   public Dimension getPreferredSize()

   {

      // size measurement code goes here

   }

   // instance fields

}
```

We display the font choices graphically by showing the actual appearance of each font (see *ListRenderingTest.java* in the demonstration code section below). In the *paintComponent* method, we display each name in its own font. We also need to make sure to match the usual

colors of the look and feel of the *JList* class. We obtain these colors by calling the *getForeground/getBackground* and *getSelectionForeground/getSelectionBackground* methods of the *JList* class. In the *getPreferredSize method*, we need to measure the size of the string, using the techniques that you saw in Volume I, Chapter 7.

Figure 13. A list box with rendered cells

To install the cell renderer, simply call the *setCellRenderer* method:

```
fontList.setCellRenderer(new FontCellRenderer());
```

Now all list cells are drawn with the custom renderer.

Actually, a simpler method for writing custom renderers works in many cases. If the rendered image just contains text, an icon, and possibly a change of color, then you can get by with configuring a *JLabel*. For example, to show the font name in its own font, we can use the following renderer:

```
class FontCellRenderer extends JLabel implements ListCellRenderer
{
   public Component getListCellRendererComponent(JList list, Object
value, int index,
      boolean isSelected, boolean cellHasFocus)
   {
      JLabel label = new JLabel();
      Font font = (Font) value;
```

```
      setText(font.getFamily());

      setFont(font);

      setOpaque(true);

      setBackground(isSelected ? list.getSelectionBackground() :
list.getBackground());

      setForeground(isSelected ? list.getSelectionForeground() :
list.getForeground());

      return this;

   }

}
```

Note that here we don't write any *paintComponent* or *getPreferredSize* methods; the *JLabel class* already implements these methods to our satisfaction. All we do is configure the label appropriately by setting its text, font, and color.

This code is a convenient shortcut for those cases in which an existing component—in this case, *JLabel*—already provides all functionality needed to render a cell value.

We could have used a *JLabel* in our sample program, but we gave you the more general code so that you can modify it when you need to do arbitrary drawings in list cells.

It is not a good idea to construct a new component in each call to *getListCellRendererComponent*. If the user scrolls through many list entries, a new component would be constructed every time. Reconfiguring an existing component is safe and much more efficient.

6.2 Demonstration of Advanced Swing

ListTest.java

```
 import java.awt.*;
  import java.awt.event.*;
  import javax.swing.*;
  import javax.swing.event.*;

  /**
  * This program demonstrates a simple fixed list of strings.
  * @version 1.23 2007-08-01
  * @author Cay Horstmann
  */
  public class ListTest
  {
  public static void main(String[] args)
  {
  EventQueue.invokeLater(new Runnable()
  {
  public void run()
  {
   JFrame frame = new ListFrame();
   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   frame.setVisible(true);
   }
   });
   }
   }
   /**
   * This frame contains a word list and a label that shows a
sentence made   up from the chosen
   * words. Note that you can select multiple words with Ctrl+click
and Shift+click.
   */
  class ListFrame extends JFrame
  {
     public ListFrame()
     {
        setTitle("ListTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        String[] words = { "quick", "brown", "hungry", "wild",
"silent", "huge", "private", "abstract", "static", "final" };

        wordList = new JList(words);
```

```
        wordList.setVisibleRowCount(4);
        JScrollPane scrollPane = new JScrollPane(wordList);

        listPanel = new JPanel();
        listPanel.add(scrollPane);
        wordList.addListSelectionListener(new ListSelectionListener()
        {
         public void valueChanged(ListSelectionEvent event)
         {
              Object[] values = wordList.getSelectedValues();

               StringBuilder text = new StringBuilder(prefix);
               for (int i = 0; i < values.length; i++)
               {
                   String word = (String) values[i];
                   text.append(word);
                   text.append(" ");
                   }
                   text.append(suffix);

                   label.setText(text.toString());
               }
           });

        buttonPanel = new JPanel();
        group = new ButtonGroup();
        makeButton("Vertical", JList.VERTICAL);
        makeButton("Vertical Wrap", JList.VERTICAL_WRAP);
        makeButton("Horizontal Wrap", JList.HORIZONTAL_WRAP);

        add(listPanel, BorderLayout.NORTH);
        label = new JLabel(prefix + suffix);
        add(label, BorderLayout.CENTER);
        add(buttonPanel, BorderLayout.SOUTH);
    }

    /**
     * Makes a radio button to set the layout orientation.
     * @param label the button label
     * @param orientation the orientation for the list
     */
    private void makeButton(String label, final int orientation)
    {
        JRadioButton button = new JRadioButton(label);
        buttonPanel.add(button);
```

```
        if (group.getButtonCount() == 0) button.setSelected(true);
        group.add(button);
        button.addActionListener(new ActionListener()
        {
              public void actionPerformed(ActionEvent event)
             {
                 wordList.setLayoutOrientation(orientation);
                 listPanel.revalidate();
              }
           });
     }

     private static final int DEFAULT_WIDTH = 400;
     private static final int DEFAULT_HEIGHT = 300;
     private JPanel listPanel;
     private JList wordList;
     private JLabel label;
     private JPanel buttonPanel;
     private ButtonGroup group;
     private String prefix = "The ";
     private String suffix = "fox jumps over the lazy dog.";}
```

javax.swing.JList 1.2

- JList(Object[] items)

constructs a list that displays these items.

- int getVisibleRowCount()
- void setVisibleRowCount(int c)

gets or sets the preferred number of rows in the list that can be displayed without a scroll bar.

- int getLayoutOrientation() 1.4
- void setLayoutOrientation(int orientation) 1.4

gets or sets the layout orientation

Parameters: orientation One of VERTICAL, VERTICAL_WRAP, HORIZONTAL_WRAP

- int getSelectionMode()
- void setSelectionMode(int mode)

gets or sets the mode that determines whether single-item or multiple-item selections are allowed.

Parameters: mode One of SINGLE_SELECTION, SINGLE_INTERVAL_SELECTION, MULTIPLE_INTERVAL_SELECTION

- void addListSelectionListener(ListSelectionListener listener)

adds to the list a listener that's notified each time a change to the selection occurs.

- Object[] getSelectedValues()

returns the selected values or an empty array if the selection is empty.

- Object getSelectedValue()

returns the first selected value or null if the selection is empty.

javax.swing.event.ListSelectionListener 1.2

- void valueChanged(ListSelectionEvent e)

is called whenever the list selection changes.

```
  import java.awt.*;

   import javax.swing.*;
   import javax.swing.event.*;
    /**
    * This program demonstrates a list that dynamically computes list
entries.
    * @version 1.23 2007-08-01
    * @author Cay Horstmann
    */
  public class LongListTest
  {
     public static void main(String[] args)
     {
    EventQueue.invokeLater(new Runnable()
           {
              public void run()
              {
                 JFrame frame = new LongListFrame();
                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
              }
           });
    }
  }

  /**
   * This frame contains a long word list and a label that shows a
sentence made up from
   * the chosen word.
   */
  class LongListFrame extends JFrame
```

```
  {
     public LongListFrame()
     {
       setTitle("LongListTest");
       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        wordList = new JList(new WordListModel(3));

wordList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        wordList.setPrototypeCellValue("www");
        JScrollPane scrollPane = new JScrollPane(wordList);

        JPanel p = new JPanel();
        p.add(scrollPane);
        wordList.addListSelectionListener(new ListSelectionListener()
           {
              public void valueChanged(ListSelectionEvent evt)
              {
                 StringBuilder word = (StringBuilder)
wordList.getSelectedValue();
                 setSubject(word.toString());
              }

           });

        Container contentPane = getContentPane();
        contentPane.add(p, BorderLayout.NORTH);
        label = new JLabel(prefix + suffix);
        contentPane.add(label, BorderLayout.CENTER);
        setSubject("fox");
     }

     /**
      * Sets the subject in the label.
      * @param word the new subject that jumps over the lazy dog
      */
     public void setSubject(String word)
     {
        StringBuilder text = new StringBuilder(prefix);
        text.append(word);
        text.append(suffix);
        label.setText(text.toString());
    }

     private static final int DEFAULT_WIDTH = 400;
     private static final int DEFAULT_HEIGHT = 300;
```

```
    private JList wordList;
    private JLabel label;
    private String prefix = "The quick brown ";
    private String suffix = " jumps over the lazy dog.";
}

/**
 * A model that dynamically generates n-letter words.
*/
class WordListModel extends AbstractListModel
{
   /**
    * Constructs the model.
    * @param n the word length
   */
   public WordListModel(int n)
  {
     length = n;
   }

   public int getSize()
   {
      return (int) Math.pow(LAST - FIRST + 1, length);
   }
  public Object getElementAt(int n)
  {
     StringBuilder r = new StringBuilder();
     ;
     for (int i = 0; i < length; i++)
     {
        char c = (char) (FIRST + n % (LAST - FIRST + 1));
        r.insert(0, c);
        n = n / (LAST - FIRST + 1);
     }
     return r;
  }
private int length;
 public static final char FIRST = 'a';
public static final char LAST = 'z';
}
```

javax.swing.JList 1.2

- JList(ListModel dataModel)

constructs a list that displays the elements in the specified model.

- Object getPrototypeCellValue()

- void setPrototypeCellValue(Object newValue)

gets or sets the prototype cell value that is used to determine the width and height of each cell in the list. The default is null, which forces the size of each cell to be measured.

- void setFixedCellWidth(int width)

if the width is greater than zero, specifies the width (in pixels) of every cell in the list. The default value is -1, which forces the size of each cell to be measured.

- void setFixedCellHeight(int height)

if the height is greater than zero, specifies the height (in pixels) of every cell in the list. The default value is -1, which forces the size of each cell to be measured.

javax.swing.ListModel 1.2

- int getSize()

returns the number of elements of the model.

- Object getElementAt(int position)

returns an element of the model at the given position.

ListRenderingTest.java

```
   import java.util.*;
   import java.awt.*;

   import javax.swing.*;
   import javax.swing.event.*;

   /**
    * This program demonstrates the use of cell renderers in a list
box.
   * @version 1.23 2007-08-01
   * @author Cay Horstmann
   */
  public class ListRenderingTest
  {
     public static void main(String[] args)
     {
        EventQueue.invokeLater(new Runnable()
           {
              public void run()
              {
                 JFrame frame = new ListRenderingFrame();
                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                 frame.setVisible(true);
              }
```

```
            });
      }
   }

   /**
    * This frame contains a list with a set of fonts and a text area
that is set to the
    * selected font.
    */
   class ListRenderingFrame extends JFrame
   {
      public ListRenderingFrame()
      {
         setTitle("ListRenderingTest");
         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

         ArrayList<Font> fonts = new ArrayList<Font>();
         final int SIZE = 24;
         fonts.add(new Font("Serif", Font.PLAIN, SIZE));
         fonts.add(new Font("SansSerif", Font.PLAIN, SIZE));
         fonts.add(new Font("Monospaced", Font.PLAIN, SIZE));
         fonts.add(new Font("Dialog", Font.PLAIN, SIZE));
         fonts.add(new Font("DialogInput", Font.PLAIN, SIZE));
         fontList = new JList(fonts.toArray());
         fontList.setVisibleRowCount(4);

fontList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
         fontList.setCellRenderer(new FontCellRenderer());
         JScrollPane scrollPane = new JScrollPane(fontList);

         JPanel p = new JPanel();
         p.add(scrollPane);
         fontList.addListSelectionListener(new ListSelectionListener()
            {
               public void valueChanged(ListSelectionEvent evt)
               {
                  Font font = (Font) fontList.getSelectedValue();
                  text.setFont(font);
               }

            });

        Container contentPane = getContentPane();
         contentPane.add(p, BorderLayout.SOUTH);
         text = new JTextArea("The quick brown fox jumps over the lazy
dog");
```

```
        text.setFont((Font) fonts.get(0));
        text.setLineWrap(true);
        text.setWrapStyleWord(true);
        contentPane.add(text, BorderLayout.CENTER);
     }

     private JTextArea text;
     private JList fontList;
    private static final int DEFAULT_WIDTH = 400;
     private static final int DEFAULT_HEIGHT = 300;
  }

  /**
   * A cell renderer for Font objects that renders the font name in
its own font.
   */
  class FontCellRenderer extends JComponent implements
ListCellRenderer
  {
     public Component getListCellRendererComponent(JList list, Object
value, int index,
           boolean isSelected, boolean cellHasFocus)
     {
        font = (Font) value;
        background = isSelected ? list.getSelectionBackground() :
list.getBackground();
        foreground = isSelected ? list.getSelectionForeground() :
list.getForeground();
        return this;
     }

     public void paintComponent(Graphics g)
     {
        String text = font.getFamily();
        FontMetrics fm = g.getFontMetrics(font);
        g.setColor(background);
        g.fillRect(0, 0, getWidth(), getHeight());
        g.setColor(foreground);
       g.setFont(font);
       g.drawString(text, 0, fm.getAscent());
    }

    public Dimension getPreferredSize()
    {
       String text = font.getFamily();
       Graphics g = getGraphics();
```

```
        FontMetrics fm = g.getFontMetrics(font);
        return new Dimension(fm.stringWidth(text), fm.getHeight());
    }

    private Font font;
    private Color background;
    private Color foreground;
}
```

javax.swing.JList 1.2

- Color getBackground()

returns the background color for unselected cells.

- Color getSelectionBackground()

returns the background color for selected cells.

- Color getForeground()

returns the foreground color for unselected cells.

- Color getSelectionForeground()

returns the foreground color for selected cells.

- void setCellRenderer(ListCellRenderer cellRenderer)

sets the renderer that paints the cells in the list.

javax.swing.ListCellRenderer 1.2

- Component getListCellRendererComponent(JList list, Object item, int index, boolean isSelected, boolean hasFocus)

returns a component whose paint method draws the cell contents. If the list cells do not have fixed size, that component must also implement getPreferredSize.

Parameters:	list	The list whose cell is being drawn
	item	The item to be drawn
	index	The index where the item is stored in the model
	isSelected	true if the specified cell was selected
	hasFocus	true if the specified cell has the focus

7 Advanced AWT

Sun Microsystems is leveraging the technology of Netscape Communications, IBM, and Lighthouse Design (now owned by Sun) to create a set of Graphical User Interface (GUI) classes

that integrate with JDK 1.1.5+, are standard with the Java ® 2 platform and provide a more polished look and feel than the standard AWT component set. The collection of APIs coming out of this effort, called the Java Foundation Classes (JFC), allows developers to build full-featured enterprise-ready applications.

JFC is composed of five APIs: AWT, Java 2D, Accessibility, Drag and Drop, and Swing. The AWT components refer to the AWT as it exists in JDK versions 1.1.2 and later. Java 2D is a graphics API based on technology licensed from IBM/Taligent. It is currently available with the Java® 2 Platform (and not usable with JDK 1.1). The Accessibility API provides assistive technologies, like screen magnifiers, for use with the various pieces of JFC. Drag and Drop support is part of the next JavaBean generation, "Glasgow," and is also available with the Java® 2 platform.

Swing includes a component set that is targeted at forms-based applications. Loosely based on Netscape's acclaimed Internet Foundation Classes (IFC), the Swing components have had the most immediate impact on Java development. They provide a set of well-groomed widgets and a framework to specify how GUIs are visually presented, independent of platform. At the time this was written, the Swing release is at 1.1 (FCS).

Though the Swing widgets were based heavily on IFC, the two APIs bear little resemblance to one another from the perspective of a developer. The look and feel of some Swing widgets and their rendering is primarily what descended from IFC, although you may notice some other commonalties.

The AWT 1.1 widgets and event model are still present for the Swing widgets. However, the 1.0 event model does not work with Swing widgets. The Swing widgets simply extend AWT by adding a new set of components, the *JComponents*, and a group of related support classes. As with AWT, Swing components are all JavaBeans and participate in the JavaBeans event model.

A subset of Swing widgets is analogous to the basic AWT widgets. In some cases, the Swing versions are simply lightweight components, rather than peer-based components. The lightweight component architecture was introduced in AWT 1.1. It allows components to exist without native operating system widgets. Instead, they participate in the Model/View/Controller (MVC) architecture, which will be described in Part II of this course. Swing also contains some new

widgets such as trees, tabbed panes, and splitter panes that will greatly improve the look and functionality of GUIs.

Swing can expand and simplify your development of cross-platform applications. The Swing collection consists of seventeen packages, each of which has its own distinct purpose. As you'll learn in this short course, these packages make it relatively easy for you to put together a variety of applications that have a high degree of sophistication and user friendliness.

javax.swing: The high level swing package primarily consists of components, adapters, default component models, and interfaces for all the delegates and models.

javax.swing.border: The border package declares the Border interface and classes, which define specific border rendering styles.

javax.swing.colorchooser: The colorchooser package contains support classes for the color chooser component.

javax.swing.event: The event package is for the Swing-specific event types and listeners. In addition to the java.awt.event types, Swing components can generate their own event types.

javax.swing.filechooser: The *filechooser* package contains support classes for the file chooser component.

javax.swing.plaf.:* The pluggable look-and-feel (PLAF) packages contain the User Interface (UI) classes (delegates) which implement the different look-and-feel aspects for Swing components. There are also PLAF packages under the *javax.swing.plaf* hierarchy.

javax.swing.table: The table package contains the support interfaces and classes the Swing table component.

javax.swing.text: The text package contains the support classes for the Swing document framework.

javax.swing.text.html.:* The text.html package contains the support classes for an HTML version 3.2 renderer and parser.

javax.swing.text.rtf: The text.rtf package contains the support classes for a basic Rich Text Format (RTF) renderer.

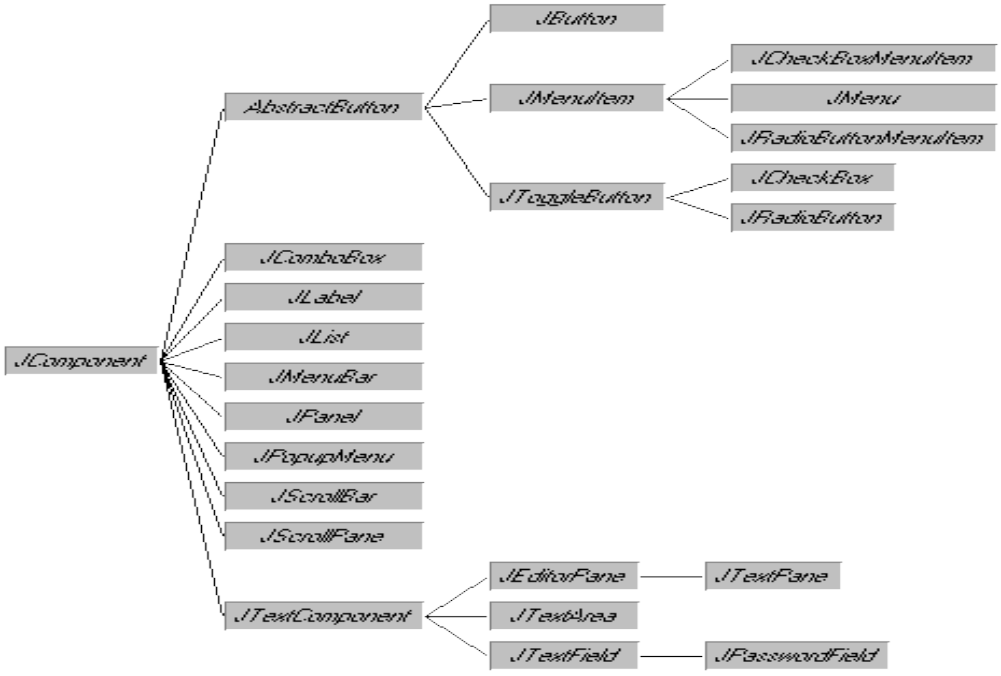
javax.swing.tree: The tree package contains the interfaces and classes which support the Swing tree component.

javax.swing.undo: The undo package provides the support classes for implementing undo/redo capabilities in a GUI.

javax.accessibility: The JFC Accessibility package is included with the Swing classes. However, its usage is not discussed here.

Widgets, Widgets, Widgets

This section describes how to use the various Swing widgets. The Swing component hierarchy is shown in two parts for comparison with AWT. Part 1 of the component hierarchy is similar to that of AWT. However, there are over twice as many components in Swing as in AWT. Part 2 shows the expanded Swing component set. This group of components appeals most to developers, as it provides a much richer set of widgets to use.



Component Hierarchy: Part 1--AWT Similar

7.1 Overview of Advanced AWT

The Rendering Pipeline

The original JDK 1.0 had a very simple mechanism for drawing shapes. You selected color and paint mode, and called methods of the Graphics class such as *drawRect* or *fillOval*. The Java 2D API supports many more options.

- You can easily produce a wide variety of shapes.
- You have control over the stroke, the pen that traces shape boundaries.
- You can fill shapes with solid colors, varying hues, and repeating patterns.
- You can use transformations to move, scale, rotate, or stretch shapes.
- You can clip shapes to restrict them to arbitrary areas.
- You can select composition rules to describe how to combine the pixels of a new shape with existing pixels.
- You can give rendering hints to make trade-offs between speed and drawing quality.

To draw a shape, you go through the following steps:

1. Obtain an object of the *Graphics2D* class. This class is a subclass of the Graphics class. Ever since Java SE 1.2, methods such as paint and *paintComponent* automatically receive an object of the *Graphics2D* class. Simply use a cast, as follows:

```
public void paintComponent(Graphics g)
{ Graphics2D g2 = (Graphics2D) g;
   . . .}
```

2. Use the *setRenderingHints* method to set rendering hints: trade-offs between speed and drawing quality.

```
RenderingHints hints = . . .;
```

```
g2.setRenderingHints(hints);
```

3. Use the *setStroke* method to set the stroke. The stroke draws the outline of the shape. You can select the thickness and choose among solid and dotted lines.

```
Stroke stroke = . . .;

g2.setStroke(stroke);
```

4. Use the *setPaint* method to set the paint. The paint fills areas such as the stroke path or the interior of a shape. You can create solid color paint, paint with changing hues, or tiled fill patterns.

```
Paint paint = . . .;

g2.setPaint(paint);
```

5. Use the clip method to set the clipping region.

```
Shape clip = . . .;

g2.clip(clip);
```

6. Use the transform method to set a transformation from user space to device space. You use transformations if it is easier for you to define your shapes in a custom coordinate system than by using pixel coordinates.

```
AffineTransform transform = . . .;

g2.transform(transform);
```

7. Use the *setComposite* method to set a composition rule that describes how to combine the new pixels with the existing pixels.

```
Composite composite = . . .;

g2.setComposite(composite);
```

8. Create a shape. The Java 2D API supplies many shape objects and methods to combine shapes.

```
Shape shape = . . .;
```

9. Draw or fill the shape. If you draw the shape, its outline is stroked. If you fill the shape, the interior is painted.

```
g2.draw(shape);
```

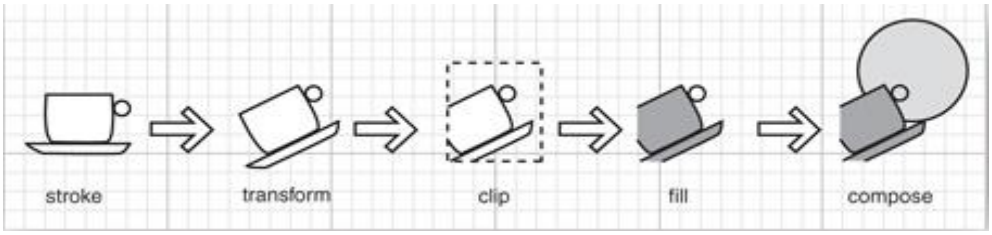
```
g2.fill(shape);
```

Of course, in many practical circumstances, you don't need all these steps. There are reasonable defaults for the settings of the 2D graphics context. You would change the settings only if you want to change the defaults.

In the following sections, you will see how to describe shapes, strokes, paints, transformations, and composition rules.

The various set methods simply set the state of the 2D graphics context. They don't cause any drawing. Similarly, when you construct Shape objects, no drawing takes place. A shape is only rendered when you call draw or fill. At that time, the new shape is computed in a rendering pipeline (see Figure 14).

Figure 14. The rendering pipeline



In the rendering pipeline, the following steps take place to render a shape:

1. The path of the shape is stroked.

2. The shape is transformed.

3. The shape is clipped. If there is no intersection between the shape and the clipping area, then the process stops.

4. The remainder of the shape after clipping is filled.

5. The pixels of the filled shape are composed with the existing pixels. (In Figure 14, the circle is part of the existing pixels, and the cup shape is superimposed over it.)

Shapes

Here are some of the methods in the Graphics class to draw shapes:

```
drawLine
drawRectangle
drawRoundRect
draw3DRect
drawPolygon
drawPolyline
drawOval
drawArc
```

There are also corresponding fill methods. These methods have been in the Graphics class ever since JDK 1.0. The Java 2D API uses a completely different, object-oriented approach. Instead of methods, there are classes:

```
Line2D
Rectangle2D
RoundRectangle2D
Ellipse2D
```

```
Arc2D

QuadCurve2D

CubicCurve2D

GeneralPath
```

These classes all implement the Shape interface.

Finally, the *Point2D* class describes a point with an x- and a y- coordinate. Points are useful to define shapes, but they aren't themselves shapes.

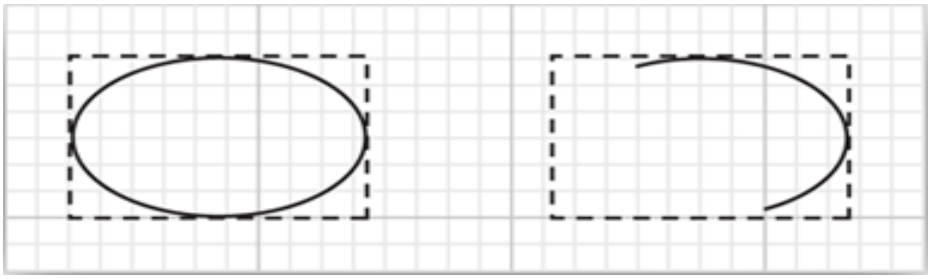
To draw a shape, you first create an object of a class that implements the Shape interface and then call the draw method of the *Graphics2D* class.

The *Line2D, Rectangle2D, RoundRectangle2D, Ellipse2D,* and *Arc2D* classes correspond to the *drawLine, drawRectangle, drawRoundRect, drawOval,* and *drawArc* methods. (The concept of a "3D rectangle" has died the death that it so richly deserved—there is no analog to the draw3DRect method.) The Java 2D API supplies two additional classes: quadratic and cubic curves. We discuss these shapes later in this section. There is no Polygon2D class. Instead, the *GeneralPath* class describes paths that are made up from lines, quadratic and cubic curves. You can use a *GeneralPath* to describe a polygon; we show you how later in this section.

```
The classes

Rectangle2D

RoundRectangle2D

Ellipse2D

Arc2D
```

all inherit from a common superclass *RectangularShape*. Admittedly, ellipses and arcs are not rectangular, but they have a bounding rectangle (see Figure 15).

Figure 15. The bounding rectangle of an ellipse and an arc



Each of the classes with a name ending in "2D" has two subclasses for specifying coordinates as float or double quantities. In Volume I, you already encountered *Rectangle2D.Float* and *Rectangle2D.Double*.

The same scheme is used for the other classes, such as *Arc2D.Float* and *Arc2D.Double*.

Internally, all graphics classes use float coordinates because float numbers use less storage space and they have sufficient precision for geometric computations. However, the Java programming language makes it a bit more tedious to manipulate float numbers. For that reason, most methods of the graphics classes use double parameters and return values. Only when constructing a 2D object must you choose between a constructor with float or double coordinates. For example,

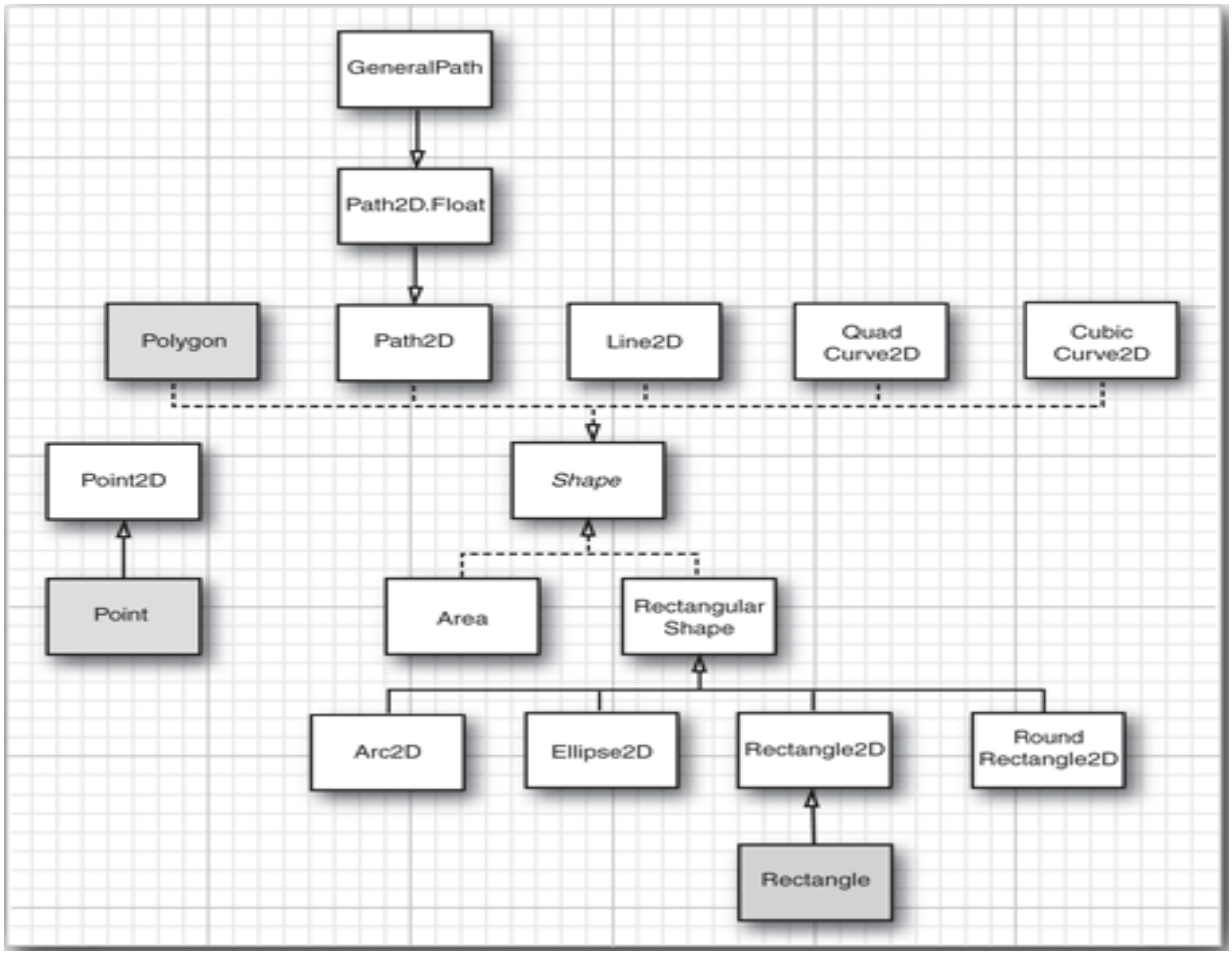
```
Rectangle2D floatRect = new Rectangle2D.Float(5F, 10F, 7.5F, 15F);
Rectangle2D doubleRect = new Rectangle2D.Double(5, 10, 7.5, 15);
```

The *Xxx2D.Float* and *Xxx2D.Double* classes are subclasses of the *Xxx2D* classes. After object construction, essentially no benefit accrues from remembering the subclass, and you can just store the constructed object in a *superclass* variable, just as in the code example.

As you can see from the curious names, the *Xxx2D.Float* and *Xxx2D.Double* classes are also inner classes of the Xxx2D classes. That is just a minor syntactical convenience, to avoid an inflation of outer class names.

Figure 16 shows the relationships between the shape classes. However, the Double and Float subclasses are omitted. Legacy classes from the pre-2D library are marked with a gray fill.

Figure 16. Relationships between the shape classes



Using the Shape Classes

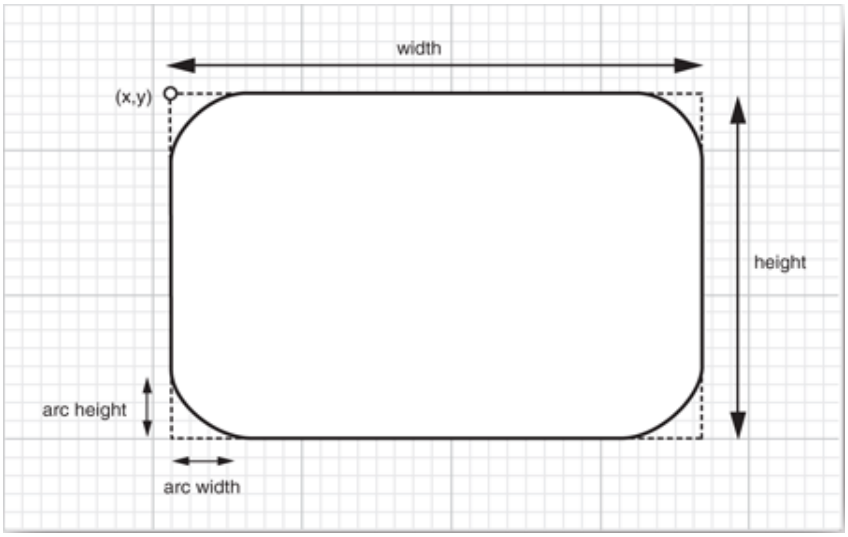
You already saw how to use the *Rectangle2D*, *Ellipse2D*, and *Line2D* classes in Volume I, Chapter 7. In this section, you will learn how to work with the remaining 2D shapes.

For the *RoundRectangle2D* shape, you specify the top-left corner, width and height, and the x- and y-dimension of the corner area that should be rounded (see Figure 17). For example, the call

```
RoundRectangle2D r = new RoundRectangle2D.Double(150, 200, 100, 50, 20, 20);
```

produces a rounded rectangle with circles of radius 20 at each of the corners.

Figure 17. Constructing a RoundRectangle2D



To construct an arc, you specify the bounding box, the start angle, the angle swept out by the arc (see Figure 18), and the closure type, one of Arc2D.OPEN, Arc2D.PIE, or Arc2D.CHORD.

```
Arc2D a = new Arc2D(x, y, width, height, startAngle, arcAngle,
closureType);
```

Figure 18. Constructing an elliptical arc

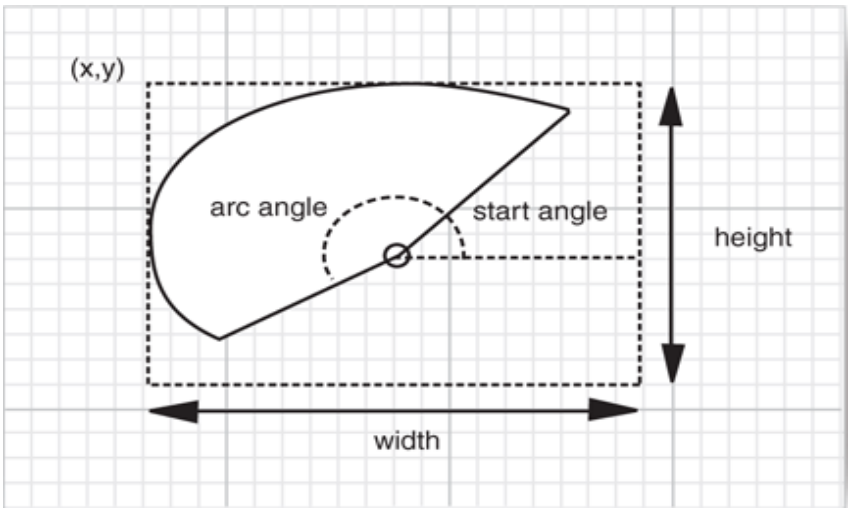
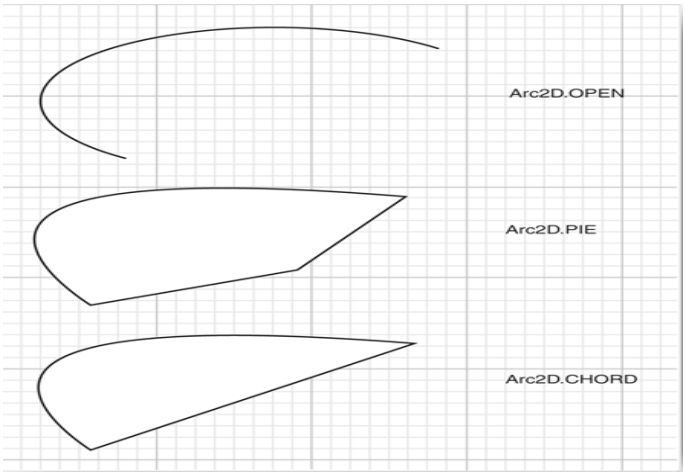


Figure 19. Arc types



If the arc is elliptical, the computation of the arc angles is not at all straightforward. The API documentation states: "The angles are specified relative to the non-square framing rectangle such that 45 degrees always falls on the line from the center of the ellipse to the upper right corner of the framing rectangle. As a result, if the framing rectangle is noticeably longer along one axis than the other, the angles to the start and end of the arc segment will be skewed farther along the longer axis of the frame." Unfortunately, the documentation is silent on how to compute this "skew." Here are the details:

Suppose the center of the arc is the origin and the point (x, y) lies on the arc. You get a skewed angle with the following formula:

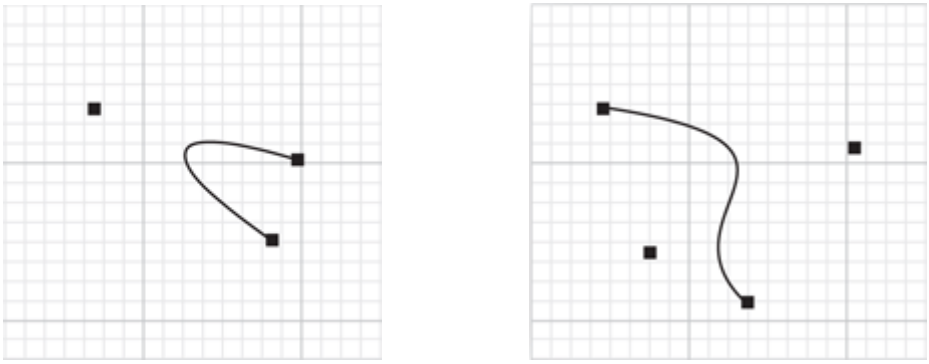
```
skewedAngle = Math.toDegrees(Math.atan2(x * width, y * height));
```

The result is a value between -180 and 180. Compute the skewed start and end angles in this way. Then, compute the difference between the two skewed angles. If the start angle or the angle difference is negative, add 360. Then, supply the start angle and the angle difference to the arc constructor.

If you run the example program at the end of this section, then you can visually check that this calculation yields the correct values for the arc constructor

The Java 2D API supports quadratic and cubic curves. In this chapter, we do not get into the mathematics of these curves. We suggest you get a feel for how the curves look by running the program in *shapetest.java*. As you can see in Figures 19, quadratic and cubic curves are specified by two end points and one or two control points. Moving the control points changes the shape of the curves.

Figure 19. A quadratic curve



To construct quadratic and cubic curves, you give the coordinates of the end points and the control points. For example,

```
QuadCurve2D q = new QuadCurve2D.Double(startX, startY, controlX, controlY,
endX, endY);
```

```
CubicCurve2D c = new CubicCurve2D.Double(startX, startY, control1X,
control1Y, control2X, control2Y, endX, endY);
```

Quadratic curves are not very flexible, and they are not commonly used in practice. Cubic curves (such as the Bezier curves drawn by the *CubicCurve2D* class) are, however, very common. By combining many cubic curves so that the slopes at the connection points match, you can create complex, smooth-looking curved shapes. For more information, we refer you to Computer Graphics: Principles and Practice, Second Edition in C by James D. Foley, Andries van Dam, Steven K. Feiner, et al. (Addison-Wesley 1995).

You can build arbitrary sequences of line segments, quadratic curves, and cubic curves, and store them in a *GeneralPath* object. You specify the first coordinate of the path with the *moveTo* method. For example,

```
GeneralPath path = new GeneralPath();
```

```
path.moveTo(10, 20);
```

You then extend the path by calling one of the methods *lineTo*, *quadTo*, or *curveTo*. These methods extend the path by a line, a quadratic curve, or a cubic curve. To call *lineTo*, supply the end point. For the two curve methods, supply the control points, then the end point. For example,

```
path.lineTo(20, 30);
path.curveTo(control1X, control1Y, control2X, control2Y, endX, endY);
```

You close the path by calling the *closePath* method. It draws a line back to the starting point of the path.

To make a polygon, simply call *moveTo* to go to the first corner point, followed by repeated calls to *lineTo* to visit the other corner points. Finally, call *closePath* to close the polygon. The program in *shapetest.java* shows this in more detail.

A general path does not have to be connected. You can call *moveTo* at any time to start a new path segment.

Finally, you can use the append method to add arbitrary Shape objects to a general path. The outline of the shape is added to the end to the path. The second parameter of the append method is true if the new shape should be connected to the last point on the path, false if it should not be connected. For example, the call

```
Rectangle2D r = . . .;
path.append(r, false);
```

appends the outline of a rectangle to the path without connecting it to the existing path. But

```
path.append(r, true);
```

adds a straight line from the end point of the path to the starting point of the rectangle, and then adds the rectangle outline to the path.

The program in *shapetest.java* lets you create sample paths. Figures 19 shows sample runs of the program. You pick a shape maker from the combo box. The program contains shape makers for

- Straight lines.
- Rectangles, round rectangles, and ellipses.

- Arcs (showing lines for the bounding rectangle and the start and end angles, in addition to the arc itself).
- Polygons (using a *GeneralPath*).
- Quadratic and cubic curves.

Use the mouse to adjust the control points. As you move them, the shape continuously repaints itself.

The program is a bit complex because it handles a multiplicity of shapes and supports dragging of the control points.

An abstract superclass *ShapeMaker* encapsulates the commonality of the shape maker classes. Each shape has a fixed number of control points that the user can move around. The *getPointCount* method returns that value. The abstract method

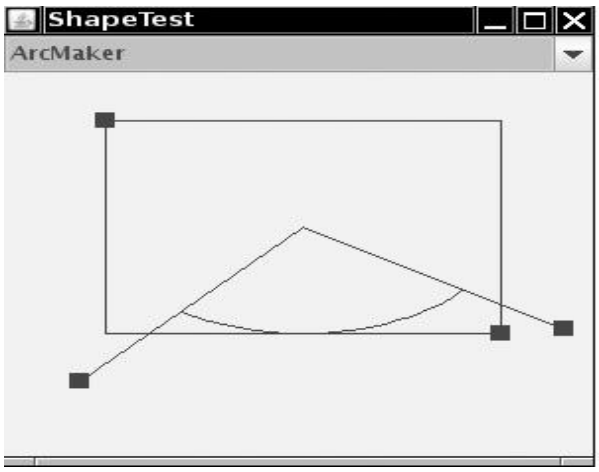
```
Shape makeShape(Point2D[] points)
```

computes the actual shape, given the current positions of the control points. The *toString* method returns the class name so that the *ShapeMaker* objects can simply be dumped into a *JComboBox*.

To enable dragging of the control points, the *ShapePanel* class handles both mouse and mouse motion events. If the mouse is pressed on top of a rectangle, subsequent mouse drags move the rectangle.

The majority of the shape maker classes are simple—their *makeShape* methods just construct and return the requested shape. However, the *ArcMaker* class needs to compute the distorted start and end angles. Furthermore, to demonstrate that the computation is indeed correct, the returned shape is a *GeneralPath* containing the arc itself, the bounding rectangle, and the lines from the center of the arc to the angle control points (see Figure 20).

Figure 20. The ShapeTest program



7.2 Demonstration of Advanced AWT

java.awt.Graphics2D 1.2

- void draw(Shape s)

draws the outline of the given shape with the current stroke.

- void fill(Shape s)

fills the interior of the given shape with the current paint.

shapetest.java

```
import java.awt.*;
 import java.awt.event.*;
 import java.awt.geom.*;
 import java.util.*;
 import javax.swing.*;

 /**
 * This program demonstrates the various 2D shapes.
 * @version 1.02 2007-08-16
* @author Cay Horstmann
*/
public class ShapeTest
{
   public static void main(String[] args)
  {
   EventQueue.invokeLater(new Runnable()
        {
```

```
         public void run()
             {
                 JFrame frame = new ShapeTestFrame();
                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
             }
          });
     }
  }

  /**
   * This frame contains a combo box to select a shape and a component
to draw it.
   */
  class ShapeTestFrame extends JFrame
  {
     public ShapeTestFrame()
   {
        setTitle("ShapeTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        final ShapeComponent comp = new ShapeComponent();
        add(comp, BorderLayout.CENTER);
        final JComboBox comboBox = new JComboBox();
        comboBox.addItem(new LineMaker());
        comboBox.addItem(new RectangleMaker());
        comboBox.addItem(new RoundRectangleMaker());
        comboBox.addItem(new EllipseMaker());
        comboBox.addItem(new ArcMaker());
        comboBox.addItem(new PolygonMaker());
        comboBox.addItem(new QuadCurveMaker());
        comboBox.addItem(new CubicCurveMaker());
        comboBox.addActionListener(new ActionListener()
           {
              public void actionPerformed(ActionEvent event)
              {
                 ShapeMaker shapeMaker = (ShapeMaker)
comboBox.getSelectedItem();
                 comp.setShapeMaker(shapeMaker);
              }
           });
        add(comboBox, BorderLayout.NORTH);
        comp.setShapeMaker((ShapeMaker) comboBox.getItemAt(0));
     }

     private static final int DEFAULT_WIDTH = 300;
```

```
     private static final int DEFAULT_HEIGHT = 300;
 }

  /**
   * This component draws a shape and allows the user to move the
points that define it.
   */
  class ShapeComponent extends JComponent
  {
     public ShapeComponent()
     {
       addMouseListener(new MouseAdapter()
          {
              public void mousePressed(MouseEvent event)
              {
                 Point p = event.getPoint();
                 for (int i = 0; i < points.length; i++)
                 {
                    double x = points[i].getX() - SIZE / 2;
                   double y = points[i].getY() - SIZE / 2;
                    Rectangle2D r = new Rectangle2D.Double(x, y, SIZE,
SIZE);
                    if (r.contains(p))
                    {
                       current = i;
                       return;
                    }
                 }
              }

              public void mouseReleased(MouseEvent event)
              {
                 current = -1;
              }
           });
        addMouseMotionListener(new MouseMotionAdapter()
          {
             public void mouseDragged(MouseEvent event)
             {
                if (current == -1) return;
               points[current] = event.getPoint();
               repaint();
             }
          });
       current = -1;
    }
```

```
 /**
  * Set a shape maker and initialize it with a random point set.
  * @param aShapeMaker a shape maker that defines a shape from a point
set
  */
 public void setShapeMaker(ShapeMaker aShapeMaker)
    {
       shapeMaker = aShapeMaker;
       int n = shapeMaker.getPointCount();
       points = new Point2D[n];
       for (int i = 0; i < n; i++)
       {
          double x = generator.nextDouble() * getWidth();
          double y = generator.nextDouble() * getHeight();
          points[i] = new Point2D.Double(x, y);
       }
       repaint();
    }
  public void paintComponent(Graphics g)
    {
       if (points == null) return;
       Graphics2D g2 = (Graphics2D) g;
       for (int i = 0; i < points.length; i++)
       {
          double x = points[i].getX() - SIZE / 2;
          double y = points[i].getY() - SIZE / 2;
          g2.fill(new Rectangle2D.Double(x, y, SIZE, SIZE));
       }

       g2.draw(shapeMaker.makeShape(points));
    }

    private Point2D[] points;
    private static Random generator = new Random();
    private static int SIZE = 10;
    private int current;
    private ShapeMaker shapeMaker;
 }
 /**
  * A shape maker can make a shape from a point set. Concrete
subclasses must return a shape
  *  in the makeShape method.
  */
 abstract class ShapeMaker
 {
```

```
    /**
     * Constructs a shape maker.
    * @param aPointCount the number of points needed to define this
shape.
     */
    public ShapeMaker(int aPointCount)
   {
      pointCount = aPointCount;
   }
    /**
     * Gets the number of points needed to define this shape.
     * @return the point count
     */
    public int getPointCount()
    {
      return pointCount;
    }
    /**
     * Makes a shape out of the given point set.
     * @param p the points that define the shape
     * @return the shape defined by the points
     */
    public abstract Shape makeShape(Point2D[] p);

    public String toString()
    {
       return getClass().getName();
   }
    private int pointCount;
 }

 /**
  * Makes a line that joins two given points.
 */
class LineMaker extends ShapeMaker
{
    public LineMaker()
    {
       super(2);
    }
 public Shape makeShape(Point2D[] p)
    {
       return new Line2D.Double(p[0], p[1]);
    }
 }
```

```
 /**
  * Makes a rectangle that joins two given corner points.
  */
 class RectangleMaker extends ShapeMaker
 {
  public RectangleMaker()
    {
       super(2);
    }
 public Shape makeShape(Point2D[] p)
    {
       Rectangle2D s = new Rectangle2D.Double();
       s.setFrameFromDiagonal(p[0], p[1]);
       return s;
    }
 }
 /**
  * Makes a round rectangle that joins two given corner points.
  */
 class RoundRectangleMaker extends ShapeMaker
 {
    public RoundRectangleMaker()
   {
       super(2);
    }
public Shape makeShape(Point2D[] p)
    {
       RoundRectangle2D s = new RoundRectangle2D.Double(0, 0, 0, 0,
20, 20);
      s.setFrameFromDiagonal(p[0], p[1]);
       return s;
    }
 }
 /**
  * Makes an ellipse contained in a bounding box with two given corner
points.
  */
 class EllipseMaker extends ShapeMaker
 {
    public EllipseMaker()
    {
       super(2);
    }
 public Shape makeShape(Point2D[] p)
    {
       Ellipse2D s = new Ellipse2D.Double();
```

```
       s.setFrameFromDiagonal(p[0], p[1]);
       return s;
    }
 }
/**
* Makes an arc contained in a bounding box with two given corner
points, and with starting
* and ending angles given by lines emanating from the center of the
bounding box and ending
*  in two given points. To show the correctness of the angle
computation, the returned
* contains the arc, the bounding box, and the lines.
*/
    class ArcMaker extends ShapeMaker
{
   public ArcMaker()
{
   super(4);
    }
public Shape makeShape(Point2D[] p)
{
double centerX = (p[0].getX() + p[1].getX()) / 2;
double centerY = (p[0].getY() + p[1].getY()) / 2;
double width = Math.abs(p[1].getX() - p[0].getX());
double height = Math.abs(p[1].getY() - p[0].getY());
    double skewedStartAngle = Math.toDegrees(Math.atan2(-(p[2].getY()
- centerY)
                                                * width, (p[2].getX() -
centerX)* height));
    double skewedEndAngle = Math.toDegrees(Math.atan2(-(p[3].getY() -
centerY)
                                               * width, (p[3].getX() -
centerX)* height));
   double skewedAngleDifference = skewedEndAngle - skewedStartAngle;
if (skewedStartAngle < 0) skewedStartAngle += 360;
           if (skewedAngleDifference < 0) skewedAngleDifference +=
360;
          Arc2D s = new Arc2D.Double(0, 0, 0, 0, skewedStartAngle,
skewedAngleDifference,
                             Arc2D.OPEN);
          s.setFrameFromDiagonal(p[0], p[1]);
    GeneralPath g = new GeneralPath();
         g.append(s, false);
         Rectangle2D r = new Rectangle2D.Double();
         r.setFrameFromDiagonal(p[0], p[1]);
              g.append(r, false);
```

```
    Point2D center = new Point2D.Double(centerX, centerY);
               g.append(new Line2D.Double(center, p[2]), false);
              g.append(new Line2D.Double(center, p[3]), false);
return g;
    }
}
/**
* Makes a polygon defined by six corner points.
*/
class PolygonMaker extends ShapeMaker
   {
           public PolygonMaker()
  {
     super(6);
}
public Shape makeShape(Point2D[] p)
{
        GeneralPath s = new GeneralPath();
        s.moveTo((float) p[0].getX(), (float) p[0].getY());
         for (int i = 1; i < p.length; i++)
         s.lineTo((float) p[i].getX(), (float) p[i].getY());
         s.closePath();
return s;
    }
}
/**
* Makes a quad curve defined by two end points and a control point.
*/
class QuadCurveMaker extends ShapeMaker
{
     public QuadCurveMaker()
{
    super(3);
}
public Shape makeShape(Point2D[] p)
{
       return new QuadCurve2D.Double(p[0].getX(), p[0].getY(),
p[1].getX(), p[1].getY(), p[2]
.getX(), p[2].getY());
     }
}
/**
* Makes a cubic curve defined by two end points and two control
points.
*/
class CubicCurveMaker extends ShapeMaker
```

```
{
     public CubicCurveMaker()
{
super(4);
}
public Shape makeShape(Point2D[] p)
    {
  return new CubicCurve2D.Double(p[0].getX(), p[0].getY(),
p[1].getX(), p[1].getY(), p[2]         .getX(), p[2].getY(),
p[3].getX(), p[3].getY());
}
 }
```

java.awt.geom.RoundRectangle2D.Double 1.2

- RoundRectangle2D.Double(double x, double y, double width, double height, double arcWidth, double arcHeight)

constructs a round rectangle with the given bounding rectangle and arc dimensions. See Figure 17 for an explanation of the arcWidth and arcHeight parameters.

java.awt.geom.Arc2D.Double 1.2

- Arc2D.Double(double x, double y, double w, double h, double startAngle, double arcAngle, int type)

constructs an arc with the given bounding rectangle, start, and arc angle and arc type. The startAngle and arcAngle are explained on page 528. The type is one of Arc2D.OPEN, Arc2D.PIE, and Arc2D.CHORD.

java.awt.geom.QuadCurve2D.Double 1.2

- QuadCurve2D.Double(double x1, double y1, double ctrlx, double ctrly, double x2, double y2)

constructs a quadratic curve from a start point, a control point, and an end point.

java.awt.geom.CubicCurve2D.Double 1.2

- CubicCurve2D.Double(double x1, double y1, double ctrlx1, double ctrly1, double ctrlx2, double ctrly2, double x2, double y2)

constructs a cubic curve from a start point, two control points, and an end point.

java.awt.geom.GeneralPath 1.2

- GeneralPath()

constructs an empty general path.

java.awt.geom.Path2D.Float 6

- void moveTo(float x, float y)

makes (x, y) the current point, that is, the starting point of the next segment.

- void lineTo(float x, float y)
- void quadTo(float ctrlx, float ctrly, float x, float y)
- void curveTo(float ctrl1x, float ctrl1y, float ctrl2x, float ctrl2y, float x, float y)

draws a line, quadratic curve, or cubic curve from the current point to the end point (x, y), and makes that end point the current point.

java.awt.geom.Path2D 6

- void append(Shape s, boolean connect)

adds the outline of the given shape to the general path. If connect is true, the current point of the general path is connected to the starting point of the added shape by a straight line.

- void closePath()

closes the path by drawing a straight line from the current point to the first point in the path.

8 JavaBeans Components

8.1 Overview of JavaBeans Components

Programmers with experience in Visual Basic will immediately know why beans are so important. Programmers coming from an environment in which the tradition is to "roll your own" for everything often find it hard to believe that Visual Basic is one of the most successful examples of reusable object technology. For those who have never worked with Visual Basic, here, in a nutshell, is how you build a Visual Basic application:

1. You build the interface by dropping components (called controls in Visual Basic) onto a form window.

2. Through property inspectors, you set properties of the components such as height, color, or other behavior.

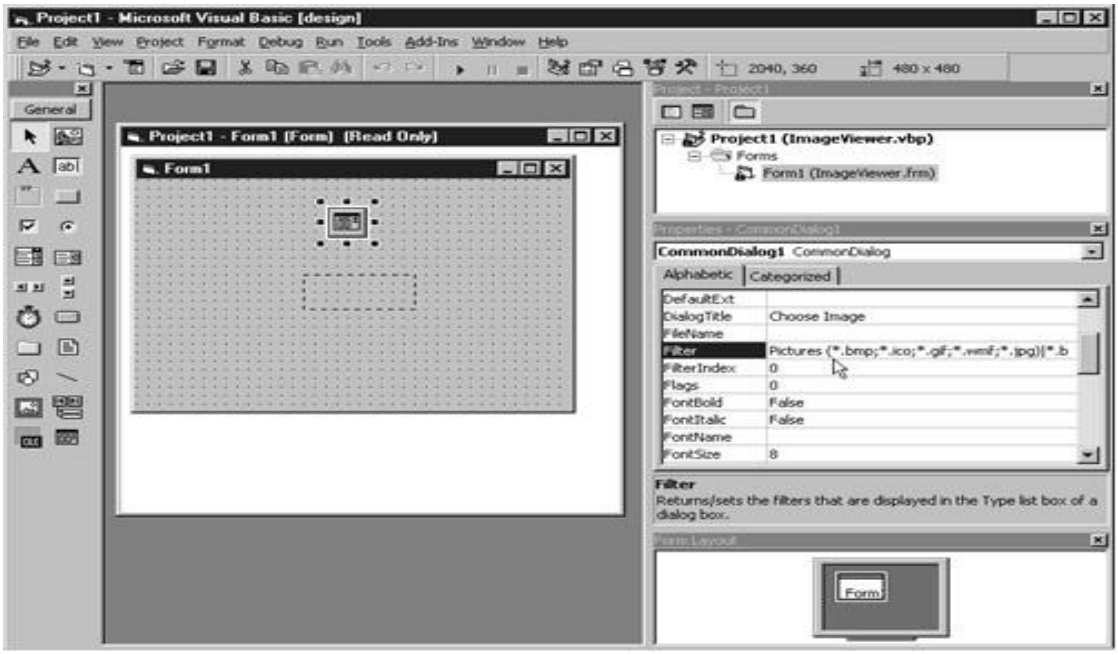
3. The property inspectors also list the events to which components can react. Some events can be hooked up through dialog boxes. For other events, you write short snippets of event handling code.

For example, in Volume I, Chapter 2, we wrote a program that displays an image in a frame. It took over a page of code. Here's what you would do in Visual Basic to create a program with pretty much the same functionality:

1. Add two controls to a window: an Image control for displaying graphics and a Common Dialog control for selecting a file.

2. Set the Filter properties of the *CommonDialog* control so that only files that the Image control can handle will show up, as shown in Figure 21.

Figure 21. The Properties window in Visual Basic for an image application



3. Write four lines of Visual Basic code that will be activated when the project first starts running. All the code you need for this sequence looks like this:

```
Private Sub Form_Load()
```

```
    CommonDialog1.ShowOpen
    Image1.Picture = LoadPicture(CommonDialog1.FileName)
End Sub
```

The code pops up the file dialog box—but only files with the right extension are shown because of how we set the filter property. After the user selects an image file, the code then tells the Image control to display it.

That's it. The layout activity, combined with these statements, gives essentially the same functionality as a page of Java code. Clearly, it is a lot easier to learn how to drop down components and set properties than it is to write a page of code.

We do not want to imply that Visual Basic is a good solution for every problem. It is clearly optimized for a particular kind of problem—UI-intensive Windows programs. The JavaBeans technology was invented to make Java technology competitive in this arena. It enables vendors to create Visual Basic-style development environments. These environments make it possible to build user interfaces with a minimum of programming.

The Bean-Writing Process

Writing a bean is not technically difficult—there are only a few new classes and interfaces for you to master. In particular, the simplest kind of bean is nothing more than a Java class that follows some fairly strict naming conventions for its methods.

ImageViewerBean.jave shows the code for an *ImageViewer* bean that could give a Java builder environment the same functionality as the Visual Basic image control we mentioned in the previous section. When you look at this code, notice that the *ImageViewerBean* class really doesn't look any different from any other class. For example, all accessor methods begin with get, and all mutator methods begin with set. As you will soon see, builder tools use this standard naming convention to discover properties. For example, *fileName* is a property of this bean because it has get and set methods.

Note that a property is not the same as an instance field. In this particular example, the *fileName* property is computed from the file instance field. Properties are conceptually at a higher level

than instance fields—they are features of the interface, whereas instance fields belong to the implementation of the class.

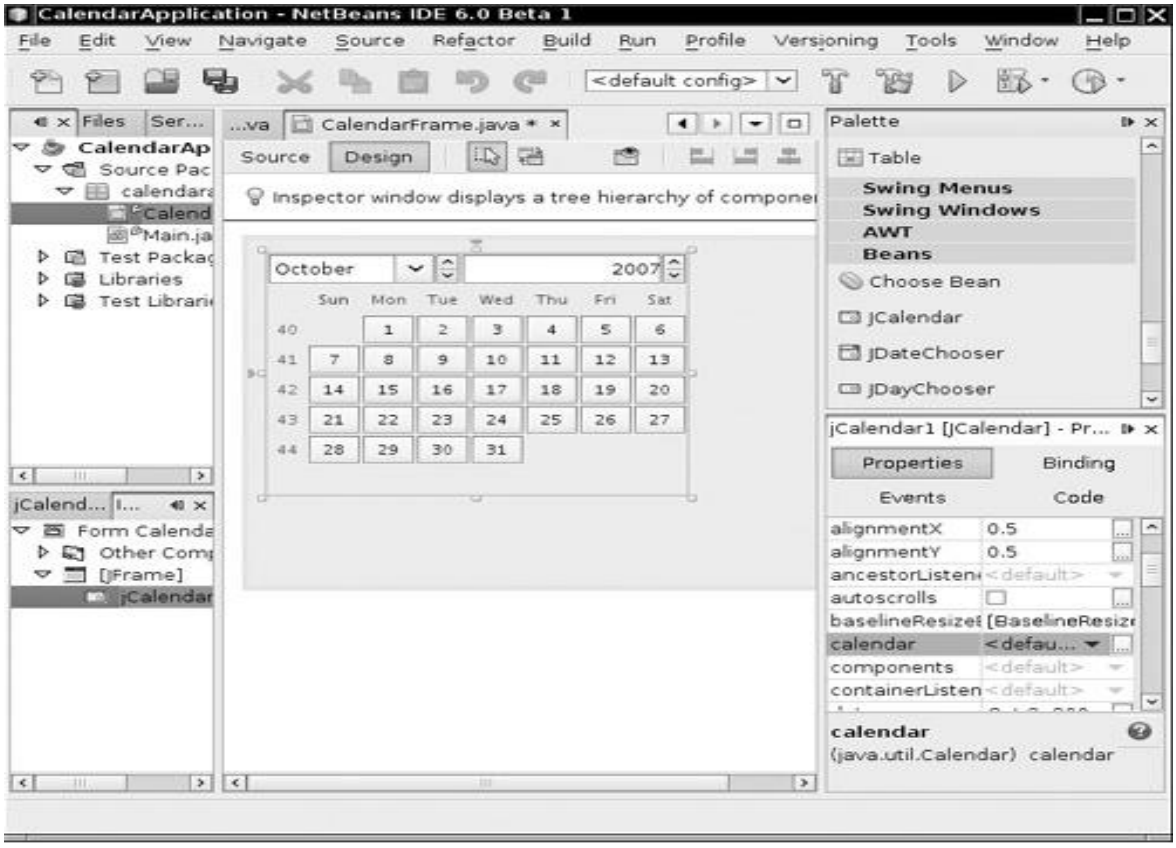
One point that you need to keep in mind when you read through the examples in this chapter is that real-world beans are much more elaborate and tedious to code than our brief examples, for two reasons.

1. Beans must be usable by less-than-expert programmers. You need to expose lots of properties so that your users can access most of the functionality of your bean with a visual design tool and without programming.

2. The same bean must be usable in a wide variety of contexts. Both the behavior and the appearance of your bean must be customizable. Again, this means exposing lots of properties.

A good example of a bean with rich behavior is *CalendarBean* by Kai Tödter (see Figure 22). The bean and its source code are freely available from http://www.toedter.com/en/jcalendar. This bean gives users a convenient way of entering dates, by locating them in a calendar display. This is obviously pretty complex and not something one would want to program from scratch. By using a bean such as this one, you can take advantage of the work of others, simply by dropping the bean into a builder tool.

Figure 22. A calendar bean



Fortunately, you need to master only a small number of concepts to write beans with a rich set of behaviors. The example beans in this chapter, although not trivial, are kept simple enough to illustrate the necessary concepts.

ImageViewerBean.java

```
package com.horstmann.corejava;
import java.awt.*;
import java.io.*;
import javax.imageio.*;
import javax.swing.*;

 /**
* A bean for viewing an image.
* @version 1.21 2001-08-15
* @author Cay Horstmann
*/
```

```
public class ImageViewerBean extends JLabel
{
public ImageViewerBean()
{
            setBorder(BorderFactory.createEtchedBorder());
}
/**
* Sets the fileName property.
* @param fileName the image file name
*/
public void setFileName(String fileName)
{
         try
{
       file = new File(fileName);
      setIcon(new ImageIcon(ImageIO.read(file)));
}
catch (IOException e)
{
       file = null;
      setIcon(null);
}
}
/**
* Gets the fileName property.
* @return the image file name
*/
public String getFileName()
{
     if (file == null) return "";
    else return file.getPath();
}
public Dimension getPreferredSize()
{
     return new Dimension(XPREFSIZE, YPREFSIZE);
}
private File file = null;
private static final int XPREFSIZE = 200;
private static final int YPREFSIZE = 200;
}
```

Using Beans to Build an Application

Before we get into the mechanics of writing beans, we want you to see how you might use or test them. *ImageViewerBean* is a perfectly usable bean, but outside a builder environment it can't show off its special features.

Each builder environment uses its own set of strategies to ease the programmer's life. We cover one environment, the *NetBeans* integrated development environment, available from http://netbeans.org.

In this example, we use two beans, *ImageViewerBean* and *FileNameBean*. You have already seen the code for *ImageViewerBean*. We will analyze the code for *FileNameBean* later in this chapter. For now, all you have to know is that clicking the button with the ". . ." label opens a file chooser.

Packaging Beans in JAR Files

To make any bean usable in a builder tool, package into a JAR file all class files that are used by the bean code. Unlike the JAR files for an applet, a JAR file for a bean needs a manifest file that specifies which class files in the archive are beans and should be included in the builder's toolbox. For example, here is the manifest file *ImageViewerBean.mf* for *ImageViewerBean.*

```
Manifest-Version: 1.0

Name: com/horstmann/corejava/ImageViewerBean.class

Java-Bean: True
```

Note the blank line between the manifest version and bean name. We place our example beans into the package com.horstmann.corejava because some builder environments have problems loading beans from the default package.

If your bean contains multiple class files, you just mention in the manifest those class files that are beans and that you want to have displayed in the toolbox. For example, you could place *ImageViewerBean* and *FileNameBean* into the same JAR file and use the manifest

```
Manifest-Version: 1.0

Name: com/horstmann/corejava/ImageViewerBean.class
```

```
Java-Bean: True

Name: com/horstmann/corejava/FileNameBean.class

Java-Bean: True
```

Some builder tools are extremely fussy about manifests. Make sure that there are no spaces after the ends of each line, that there are blank lines after the version and between bean entries, and that the last line ends in a newline.

To make the JAR file, follow these steps:

1. Edit the manifest file.
2. Gather all needed class files in a directory.
3. Run the *jar* tool as follows:

   ```
   jar cvfm JarFile ManifestFile ClassFiles
   ```

   For example,

   ```
   jar cvfm ImageViewerBean.jar ImageViewerBean.mf
   com/horstmann/corejava/*.class
   ```

Builder environments have a mechanism for adding new beans, typically by loading JAR files. Here is what you do to import beans into *NetBeans* version 6.

Compile the *ImageViewerBean* and *FileNameBean* classes and package them into JAR files. Then start *NetBeans* and follow these steps.

1. Select Tools -> Palette -> Swing/AWT Components from the menu.
2. Click the Add from JAR button.

3. In the file dialog box, move to the *ImageViewerBean* directory and select *ImageViewerBean.jar*.

4. Now a dialog box pops up that lists all the beans that were found in the JAR file. Select *ImageViewerBean*.

5. Finally, you are asked into which palette you want to place the beans. Select Beans. (There are other palettes for Swing components, AWT components, and so on.)

6. Have a look at the Beans palette. It now contains an icon representing the new bean However, the icon is just a default icon—you will see later how to add icons to a bean.

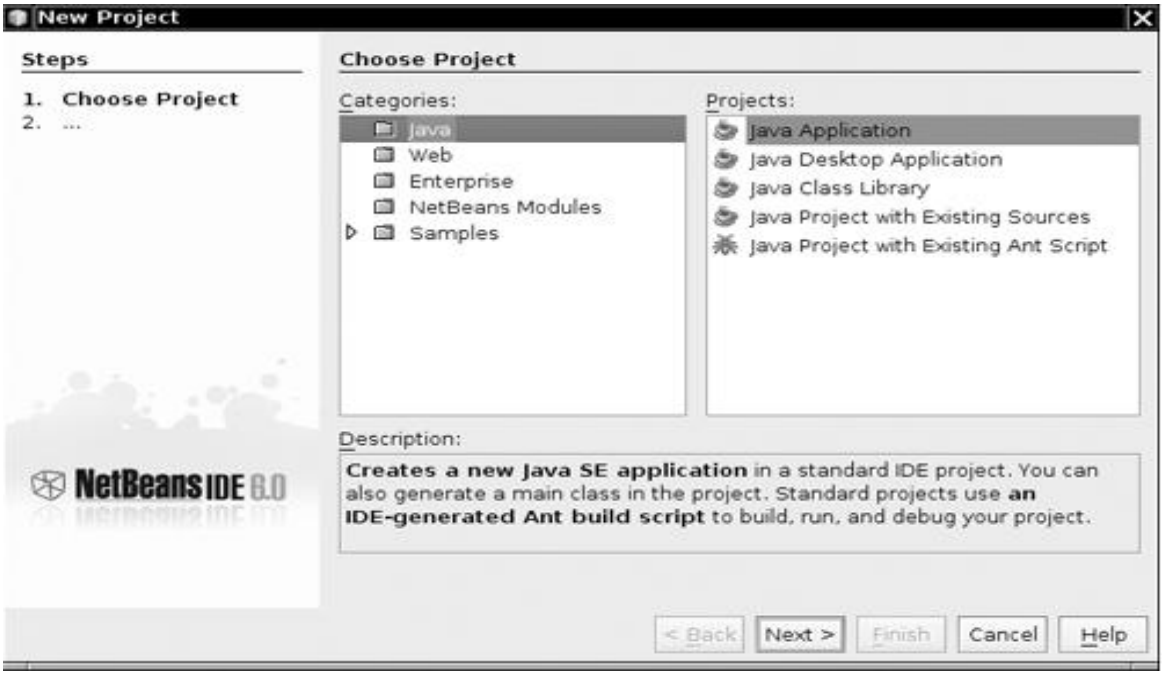
Repeat these steps with *FileNameBean*. Now you are ready to compose these beans into an application.

Composing Beans in a Builder Environment

The promise of component-based development is to compose your application from prefabricated components, with a minimum of programming. In this section, you will see how to compose an application from the *ImageViewerBean* and *FileNameBean* components.

In *NetBeans* 6, select File -> New Project from the menu. A dialog box pops up. Select Java, then Java Application (see Figure 23).

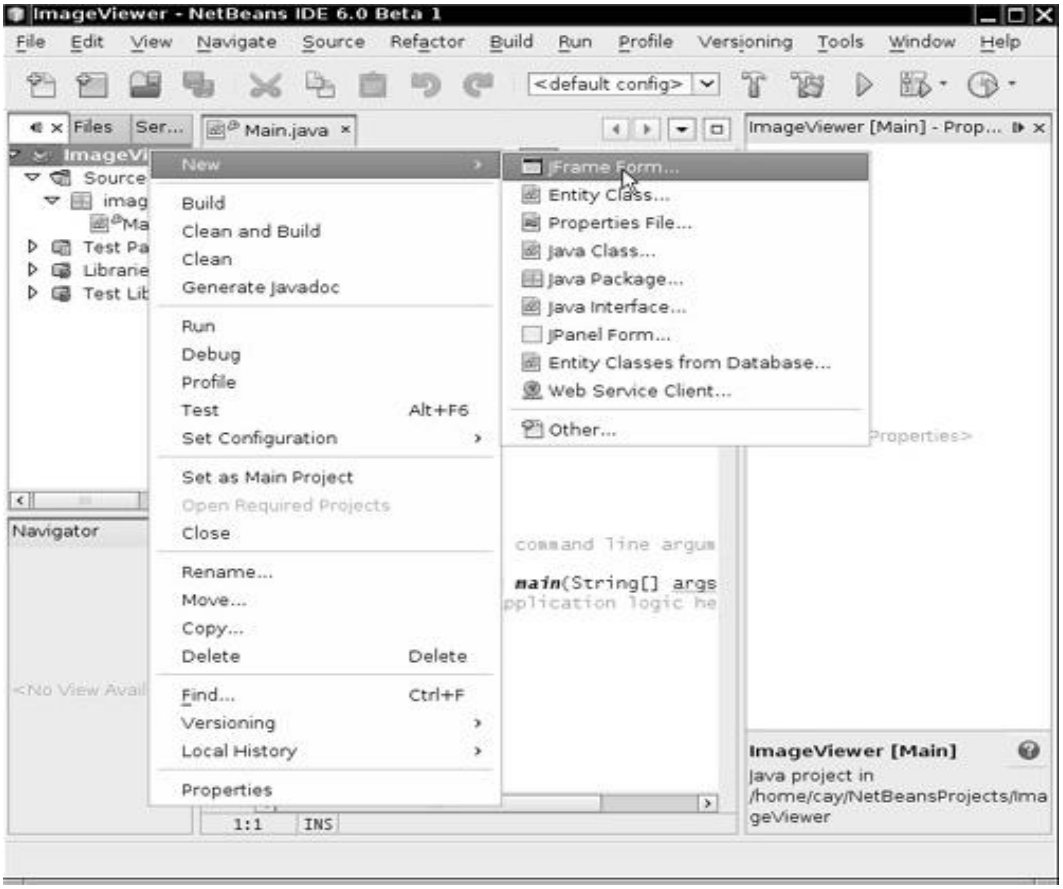
Figure 23. Creating a new project



Click the Next button. On the following screen, set a name for your application (such as *ImageViewer*), and click the Finish button. Now you see a project viewer on the left and the source code editor in the middle.

Right-click the project name in the project viewer and select New -> *JFrame* Form from the menu (see Figure 24).

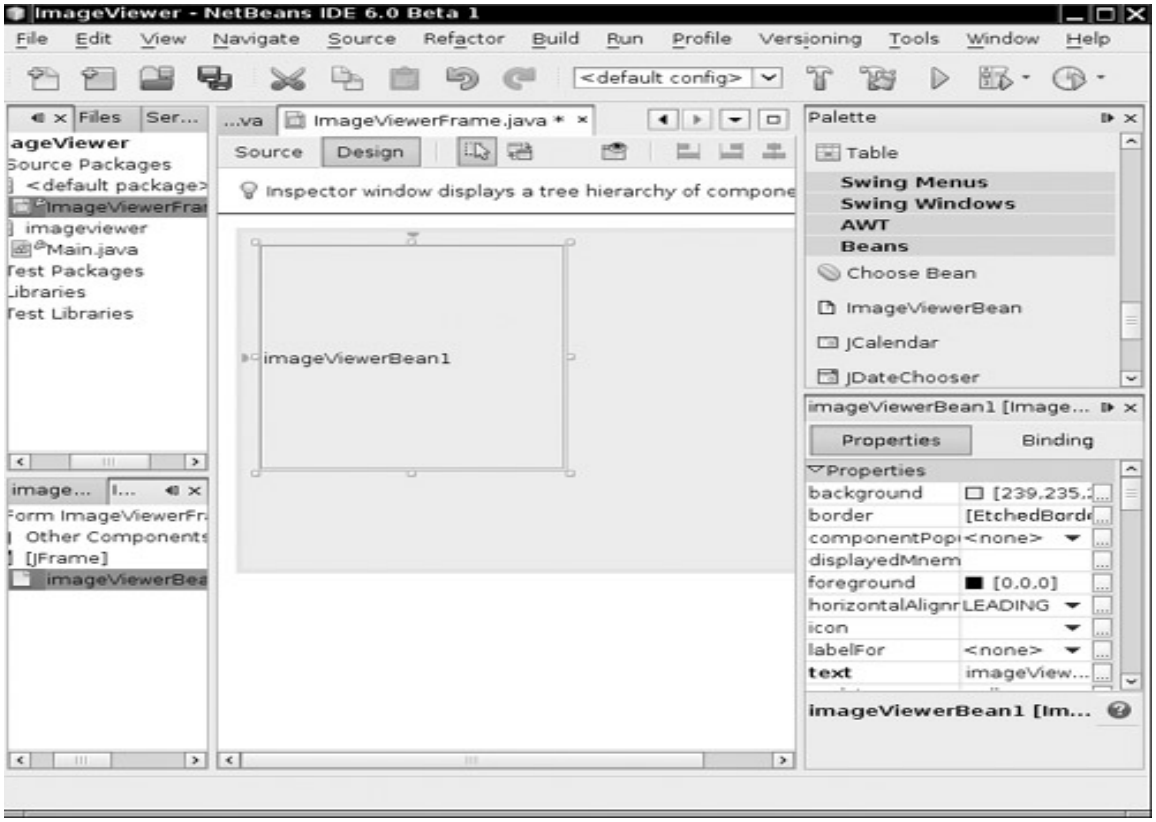
Figure 24. Creating a form view



A dialog box pops up. Enter a name for the frame class (such as *ImageViewerFrame*), and click the Finish button. You now get a form editor with a blank frame. To add a bean to the form, select the bean in the palette that is located to the right of the form editor. Then click the frame.

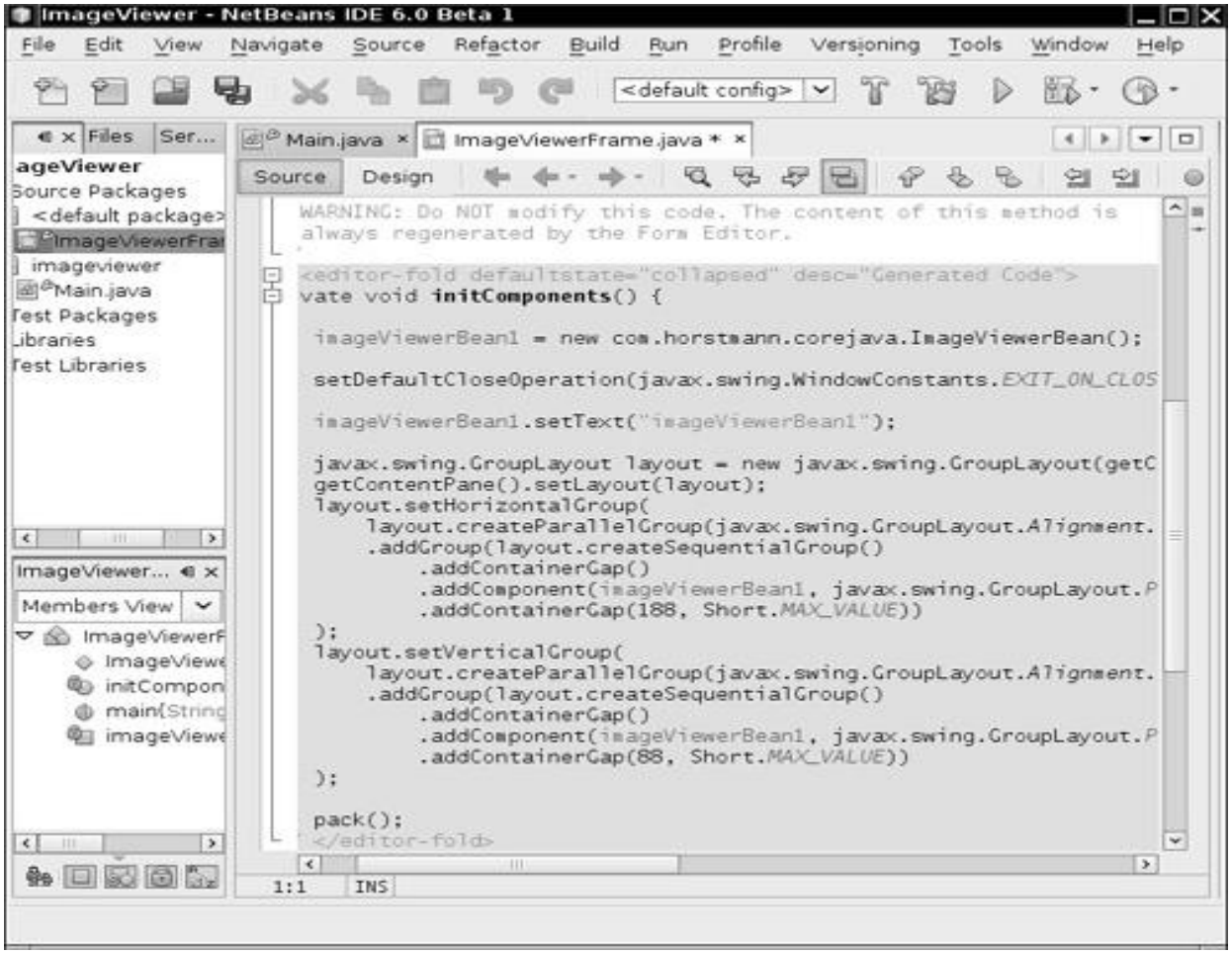
Figure 25 shows the result of adding an *ImageViewerBean* onto the frame.

Figure 25. Adding a bean



If you look into the source window, you will find that the source code now contains the Java instructions to add the bean objects to the frame (see Figure 26). The source code is bracketed by dire warnings that you should not edit it. Any edits would be lost when the builder environment updates the code as you modify the form.

Figure 26. The source code for adding the bean



A builder environment is not required to update source code as you build an application. A builder environment can generate source code when you are done editing, serialize the beans you customized, or perhaps produce an entirely different description of your building activity.

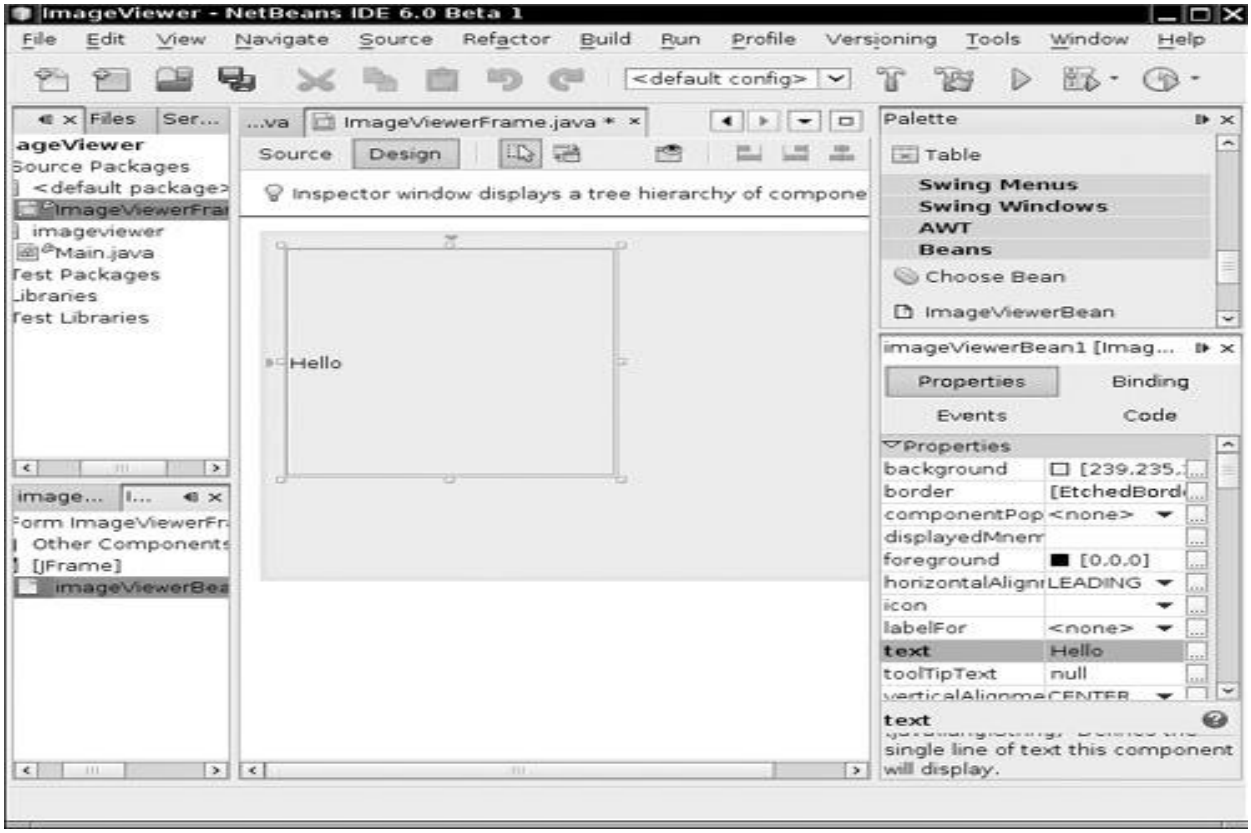
For example, the experimental Bean Builder at http://bean-builder.dev.java.net lets you design GUI applications without writing any source code at all.

The JavaBeans mechanism doesn't attempt to force an implementation strategy on a builder tool. Instead, it aims to supply information about beans to builder tools that can choose to take advantage of the information in one way or another.

Now go back to the design view and click *ImageViewerBean* in the form. On the right-hand side is a property inspector that lists the bean property names and their current values. This is a vital part of component-based development tools because setting properties at design time is how you set the initial state of a component.

For example, you can modify the text property of the label used for the image bean by simply typing a new name into the property inspector. Changing the text property is simple—you just edit a string in a text field. Try it out—set the label text to "Hello". The form is immediately updated to reflect your change (see Figure 27).

Figure 27. Changing a property in the property inspector



When you change the setting of a property, the NetBeans environment updates the source code to reflect your action. For example, if you set the text field to Hello, the instruction

```
imageViewerBean.setText("Hello");
```

is added to the *initComponents* method. As already mentioned, other builder tools might have different strategies for recording property settings.

Properties don't have to be strings; they can be values of any Java type. To make it possible for users to set values for properties of any type, builder tools use specialized property editors. (Property editors either come with the builder or are supplied by the bean developer. You see how to write your own property editors later in this chapter.)

To see a simple property editor at work, look at the foreground property. The property type is Color. You can see the color editor, with a text field containing a string [0,0,0] and a button labeled ". . ." that brings up a color chooser. Go ahead and change the foreground color. Notice that you'll immediately see the change to the property value—the label text changes color.

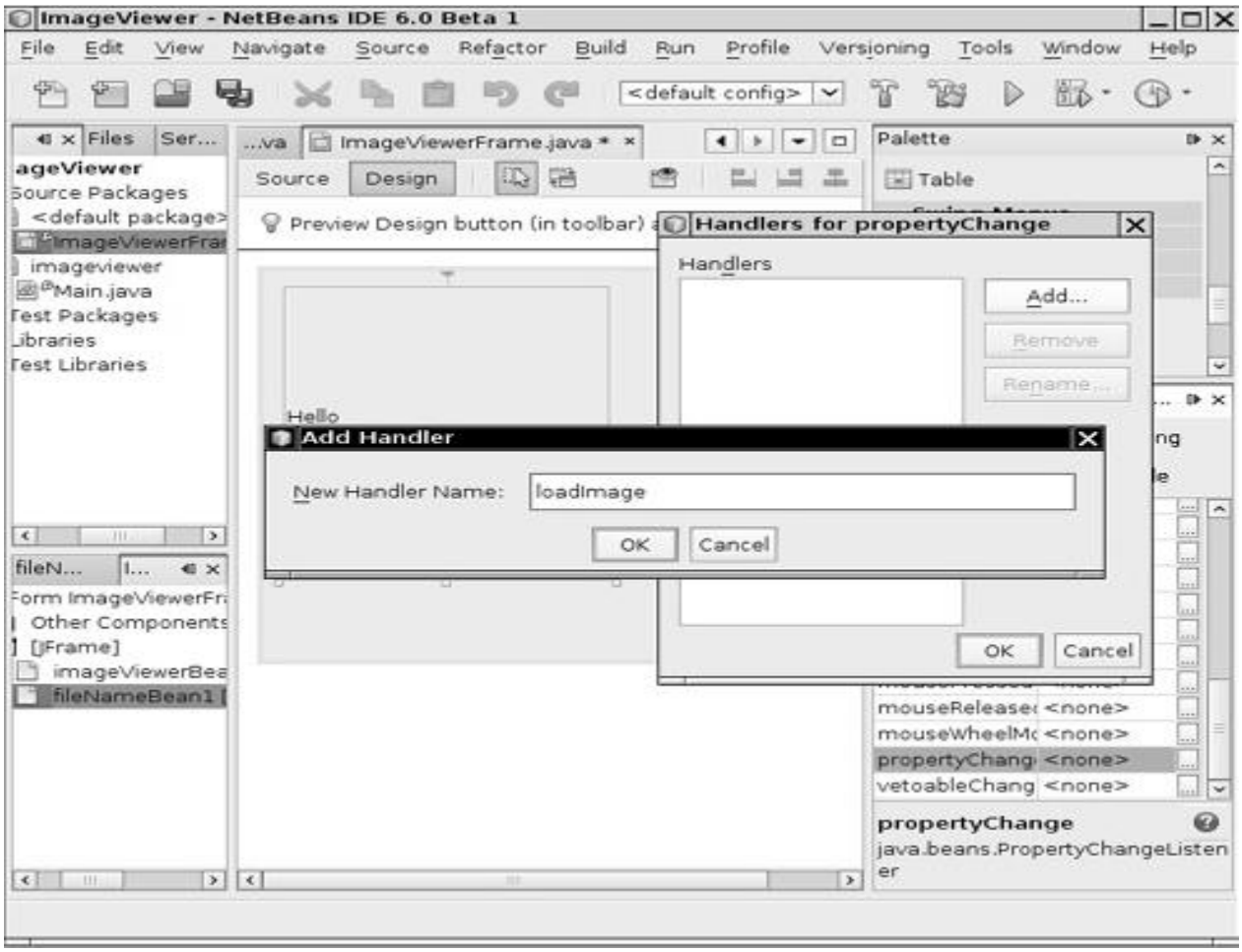
More interestingly, choose a file name for an image file in the property inspector. Once you do so, *ImageViewerBean* automatically displays the image.

If you look closely at the property inspector in NetBeans, you will find a large number of mysterious properties such as *focusCycleRoot* and *paintingForPrint*. These are inherited from the *JLabel* superclass. You will see later in this chapter how you can suppress them from the property inspector.

To complete our application, place a *FileNameBean* object into the frame. Now we want the image to be loaded when the *fileName* property of *FileNameBean* is changed. This happens through a *PropertyChange* event; we discuss these kinds of events later in this chapter.

To react to the event, select *FileNameBean* and select the Events tab from its property inspector. Then click the "..." button next to the *propertyChange* entry. A dialog box appears that shows that no handlers are currently associated with this event. Click the Add button in the dialog box. You are prompted for a method name (see Figure 28). Type *loadImage*.

Figure 28. Adding an event to a bean



Now look at the code editor. Event handling code has been added, and there is a new method:

```
private void loadImage(java.beans.PropertyChange evt)
{
   // TODO add your handling code here
}
```

Add the following line of code to that method:

```
imageViewerBean1.setFileName(fileNameBean1.getFileName());
```

Then compile and execute the frame class. You now have a complete image viewer application. Click the button with the ". . ." label and select an image file. The image is displayed in the image viewer (see Figure 29).

Figure 29. The image viewer application

Naming Patterns for Bean Properties and Events

In this section, we cover the basic rules for designing your own beans. First, we want to stress there is no cosmic beans class that you extend to build your beans. Visual beans directly or indirectly extend the Component class, but *nonvisual* beans don't have to extend any particular superclass. Remember, a bean is simply any class that can be manipulated in a builder tool. The builder tool does not look at the superclass to determine the bean nature of a class, but it analyzes the names of its methods. To enable this analysis, the method names for beans must follow certain patterns.

There is a *java.beans.Beans* class, but all methods in it are static. Extending it would, therefore, be rather pointless, even though you will see it done occasionally, supposedly for greater "clarity." Clearly, because a bean can't extend both Beans and Component, this approach can't work for visual beans. In fact, the Beans class contains methods that are designed to be called by builder tools, for example, to check whether the tool is operating at design time or run time.

Other languages for visual design environments, such as Visual Basic and C#, have special keywords such as "Property" and "Event" to express these concepts directly. The designers of the

Java specification decided not to add keywords to the language to support visual programming. Therefore, they needed an alternative so that a builder tool could analyze a bean to learn its properties or events. Actually, there are two alternative mechanisms. If the bean writer uses standard naming patterns for properties and events, then the builder tool can use the reflection mechanism to understand what properties and events the bean is supposed to expose. Alternatively, the bean writer can supply a bean information class that tells the builder tool about the properties and events of the bean. We start out using the naming patterns because they are easy to use. You'll see later in this chapter how to supply a bean information class.

The naming pattern for properties is simple: Any pair of methods

```
public Type getPropertyName()
public void setPropertyName(Type newValue)
```

corresponds to a read/write property.

For example, in our *ImageViewerBean*, there is only one read/write property (for the file name to be viewed), with the following methods:

```
public String getFileName()
public void setFileName(String newValue)
```

If you have a get method but not an associated set method, you define a read-only property. Conversely, a set method without an associated get method defines a write-only property.

The *get* and *set* methods you create can do more than simply get and set a private data field. Like any Java method, they can carry out arbitrary actions. For example, the *setFileName* method of the *ImageViewerBean* class not only sets the value of the *fileNam*e data field, but also opens the file and loads the image.

In Visual Basic and C#, properties also come from get and set methods. However, in both these languages, you explicitly define properties rather than having builder tools second-guess the programmer's intentions by analyzing method names. In those languages, properties have another advantage: Using a property name on the left side of an assignment automatically calls the set method. Using a property name in an

expression automatically calls the get method. For example, in Visual Basic you can write

```
imageBean.fileName = "corejava.gif"
```

instead of

```
imageBean.setFileName("corejava.gif");
```

This syntax was considered for Java, but the language designers felt that it was a poor idea to hide a method call behind syntax that looks like field access.

There is one exception to the get/set naming pattern. Properties that have boolean values should use an is/set naming pattern, as in the following examples:

```
public boolean isPropertyName()
public void setPropertyName(boolean b)
```

For example, an animation might have a property running, with two methods

```
public boolean isRunning()
public void setRunning(boolean b)
```

The *setRunning* method would start and stop the animation. The *isRunning* method would report its current status.

It is legal to use a get prefix for a boolean property accessor (such as *getRunning*), but the is prefix is preferred.

Be careful with the capitalization pattern you use for your method names. The designers of the JavaBeans specification decided that the name of the property in our example would be *fileName*, with a lowercase f, even though the get and set methods contain an uppercase F (*getFileName, setFileName*). The bean analyzer performs a process called *decapitalization* to derive the property name. (That is, the first character after get or set is converted to lower case.) The rationale is that this process results in method and property names that are more natural to programmers.

However, if the first two letters are upper case (such as in *getURL*), then the first letter of the property is not changed to lower case. After all, a property name of *uRL* would look ridiculous.

What do you do if your class has a pair of get and set methods that doesn't correspond to a property that you want users to manipulate in a property inspector? In your own classes, you can of course avoid that situation by renaming your methods. However, if you extend another class, then you inherit the method names from the *superclass*. This happens, for example, when your bean extends *JPanel* or *JLabel*—a large number of uninteresting properties show up in the property inspector. You will see later in this chapter how you can override the automatic property discovery process by supplying bean information. In the bean information, you can specify exactly which properties your bean should expose.

For events, the naming patterns are equally simple. A bean builder environment will infer that your bean generates events when you supply methods to add and remove event listeners. All event class names must end in Event, and the classes must extend the *EventObject* class.

Suppose your bean generates events of type *EventNameEvent*. The listener interface must be called *EventNameListener*, and the methods to manage the listeners must be called

```
public void addEventNameListener(EventNameListener e)
public void removeEventNameListener(EventNameListener e)
public EventNameListener getEventNameListeners()
```

If you look at the code for *ImageViewerBean*, you'll see that it has no events to expose. However, many Swing components generate events, and they follow this pattern. For example, the *AbstractButton* class generates *ActionEvent* objects, and it has the following methods to manage *ActionListener* objects:

```
public void addActionListener(ActionListener e)
public void removeActionListener(ActionListener e)
ActionListener[] getActionListeners()
```

If your event class doesn't extend *EventObject*, chances are that your code will compile just fine because none of the methods of the *EventObject* class are actually needed. However, your bean will mysteriously fail—the introspection mechanism will not recognize the events.

Bean Property Types

A sophisticated bean will expose lots of different properties and events. Properties can be as simple as the *fileName* property that you saw in *ImageViewerBean* and *FileNameBean* or as sophisticated as a color value or even an array of data points—we encounter both of these cases later in this chapter. The JavaBeans specification allows four types of properties, which we illustrate by various examples.

Simple Properties

A simple property is one that takes a single value such as a string or a number. The *fileName* property of the *ImageViewer* is an example of a simple property. Simple properties are easy to program: Just use the set/get naming convention we indicated earlier. For example, if you look at the code in Listing 8-1, you can see that all it took to implement a simple string property is the following:

```
public void setFileName(String f)
{
   fileName = f;
   image = . . .
   repaint();
}
public String getFileName()
{
   if (file == null) return "";
   else return file.getPath();
}
```

Indexed Properties

An indexed property specifies an array. With an indexed property, you supply two pairs of get and set methods: one for the array and one for individual entries. They must follow this pattern:

```
Type[] getPropertyName()

void setPropertyName(Type[] newValue)

Type getPropertyName(int i)

void setPropertyName(int i, Type newValue)
```

For example, the *FileNameBean* uses an indexed property for the file extensions. It provides these four methods:

```
public String[] getExtensions() { return extensions; }

public void setExtensions(String[] newValue) { extensions = newValue;
}

public String getExtensions(int i)

{

   if (0 <= i && i < extensions.length) return extensions[i];

   else return "";

}

public void setExtensions(int i, String newValue)

{

   if (0 <= i && i < extensions.length) extensions[i] = value;

}

. . .

private String[] extensions;
```

The *setPropertyName(int, Type)* method cannot be used to grow the array. To grow the array, you must manually build a new array and then pass it to the *setPropertyName(Type[])* method.

Bound Properties

Bound properties tell interested listeners that their value has changed. For example, the *fileName* property in *FileNameBean* is a bound property. When the file name changes, then *ImageViewerBean* is automatically notified and it loads the new file.

To implement a bound property, you must implement two mechanisms:

1. Whenever the value of the property changes, the bean must send a *PropertyChange* event to all registered listeners. This change can occur when the set method is called or when some other method (such as the action listener of the "..." button) changes the value.
2. To enable interested listeners to register themselves, the bean has to implement the following two methods:
3. void *addPropertyChangeListener(PropertyChangeListener listener)*

```
void removePropertyChangeListener(PropertyChangeListener listener)
```

It is also recommended (but not required) to provide the method

PropertyChangeListener[] getPropertyChangeListeners()

The *java.beans* package has a convenience class, called *PropertyChangeSupport*, that manages the listeners for you. To use this convenience class, add an instance field of this class:

```
private PropertyChangeSupport changeSupport = new
PropertyChangeSupport(this);
```

Delegate the task of adding and removing property change listeners to that object.

```
public void addPropertyChangeListener(PropertyChangeListener listener)
{
```

```
   changeSupport.addPropertyChangeListener(listener);
}
public void removePropertyChangeListener(PropertyChangeListener
listener)
{
   changeSupport.removePropertyChangeListener(listener);
}
public PropertyChangeListener[] getPropertyChangeListeners()
{
   return changeSupport.getPropertyChangeListeners();
}
```

Whenever the value of the property changes, use the *firePropertyChange* method of the *PropertyChangeSupport* object to deliver an event to all the registered listeners. That method has three parameters: the name of the property, the old value, and the new value. Here is the boilerplate code for a typical setter of a bound property:

```
public void setValue(Type newValue)
{
   Type oldValue = getValue();
   value = newValue;
   changeSupport.firePropertyChange("propertyName", oldValue,
newValue);
}
```

To fire a change of an indexed property, you call

```
changeSupport.fireIndexedPropertyChange("propertyName", index,
oldValue, newValue);
```

If your bean extends any class that ultimately extends the Component class, then you

do not need to implement the *addPropertyChangeListener, removePropertyChangeListener, and getPropertyChangeListeners* methods. These methods are already implemented in the Component superclass. To notify the listeners of a property change, simply call the *firePropertyChange* method of the *JComponent* superclass. Unfortunately, firing of indexed property changes is not supported.

Other beans that want to be notified when the property value changes must add a *PropertyChangeListener*. That interface contains only one method:

```
void propertyChange(PropertyChangeEvent event)
```

The *PropertyChangeEvent* object holds the name of the property and the old and new values, obtainable with the *getPropertyName, getOldValue, and getNewValue* methods.

If the property type is not a class type, then the property value objects are instances of the usual wrapper classes.

Constrained Properties

A constrained property is constrained by the fact that any listener can "veto" proposed changes, forcing it to revert to the old setting. The Java library contains only a few examples of constrained properties. One of them is the closed property of the *JInternalFrame* class. If someone tries to call *setClosed(true)* on an internal frame, then all of its *VetoableChangeListeners* are notified. If any of them throws a *PropertyVetoException*, then the closed property is not changed, and the *setClosed* method throws the same exception. In particular, a *VetoableChangeListener* may veto closing the frame if its contents have not been saved.

To build a constrained property, your bean must have the following two methods to manage *VetoableChangeListener* objects:

```
public void addVetoableChangeListener(VetoableChangeListener
listener);

public void removeVetoableChangeListener(VetoableChangeListener
listener);
```

It also should have a method for getting all listeners:

```
VetoableChangeListener[] getVetoableChangeListeners()
```

Just as there is a convenience class to manage property change listeners, there is a convenience class, called *VetoableChangeSupport*, that manages vetoable change listeners. Your bean should contain an object of this class.

```
private VetoableChangeSupport vetoSupport = new
VetoableChangeSupport(this);
```

Adding and removing listeners should be delegated to this object. For example:

```
public void addVetoableChangeListener(VetoableChangeListener listener)
{
   vetoSupport.addVetoableChangeListener(listener);
}
public void removeVetoableChangeListener(VetoableChangeListener
listener)
{
   vetoSupport.removeVetoableChangeListener(listener);
}
```

To update a constrained property value, a bean uses the following three-phase approach:

1. Notify all vetoable change listeners of the intent to change the property value. (Use the *fireVetoableChange* method of the *VetoableChangeSupport* class.)
2. If none of the vetoable change listeners has thrown a *PropertyVetoException*, then update the value of the property.
3. Notify all property change listeners to confirm that a change has occurred.

For example,

```
public void setValue(Type newValue) throws PropertyVetoException
{
```

```
    Type oldValue = getValue();
    vetoSupport.fireVetoableChange("value", oldValue, newValue);
    // survived, therefore no veto
    value = newValue;
    changeSupport.firePropertyChange("value", oldValue, newValue);
}
```

It is important that you don't change the property value until all the registered vetoable change listeners have agreed to the proposed change. Conversely, a vetoable change listener should never assume that a change that it agrees to is actually happening. The only reliable way to get notified when a change is actually happening is through a property change listener.

BeanInfo Classes

If you use the standard naming patterns for the methods of your bean class, then a builder tool can use reflection to determine features such as properties and events. This process makes it simple to get started with bean programming, but naming patterns are rather limiting. As your beans become complex, there might be features of your bean that naming patterns will not reveal. Moreover, as we already mentioned, many beans have get/set method pairs that should not correspond to bean properties.

If you need a more flexible mechanism for describing information about your bean, define an object that implements the *BeanInfo* interface. When you provide such an object, a builder tool will consult it about the features that your bean supports.

The name of the bean info class must be formed by adding BeanInfo to the name of the bean. For example, the bean info class associated to the class *ImageViewerBean* must be named *ImageViewerBeanBeanInfo*. The bean info class must be part of the same package as the bean itself.

You won't normally write a class that implements all methods of the BeanInfo interface. Instead, you should extend the *SimpleBeanInfo* convenience class that has default implementations for all the methods in the BeanInfo interface.

The most common reason for supplying a BeanInfo class is to gain control of the bean properties. You construct a *PropertyDescriptor* for each property by supplying the name of the property and the class of the bean that contains it.

```
PropertyDescriptor descriptor = new PropertyDescriptor("fileName",
ImageViewerBean.class);
```

Then implement the *getPropertyDescriptors* method of your BeanInfo class to return an array of all property descriptors.

For example, suppose *ImageViewerBean* wants to hide all properties that it inherits from the *JLabel* superclass and expose only the fileName property. The following *BeanInfo* class does just that:

```
// bean info class for ImageViewerBean

class ImageViewerBeanBeanInfo extends SimpleBeanInfo

{

   public PropertyDescriptor[] getPropertyDescriptors()

   {

      return propertyDescriptors;

   }

   private PropertyDescriptor[] propertyDescriptors =  new
PropertyDescriptor[]

      {

         new PropertyDescriptor("fileName", ImageViewerBean.class);

      };

}
```

Other methods also return *EventSetDescriptor* and *MethodDescriptor* arrays, but they are less commonly used. If one of these methods returns null (as is the case for the *SimpleBeanInfo* methods), then the standard naming patterns apply. However, if you override a method to return a non-null array, then you must include all properties, events, or methods in your array.

Sometimes, you might want to write generic code that discovers properties or events of an arbitrary bean. Call the static *getBeanInfo* method of the Introspector class. The Introspector constructs a BeanInfo class that completely describes the bean, taking into account the information in *BeanInfo* companion classes.

Another useful method in the *BeanInfo* interface is the *getIcon* method that lets you give your bean a custom icon. Builder tools will display the icon in a palette. Actually, you can specify four separate icon bitmaps. The *BeanInfo* interface has four constants that cover the standard sizes:

ICON_COLOR_16x16

ICON_COLOR_32x32

ICON_MONO_16x16

ICON_MONO_32x32

In the following class, we use the loadImage convenience method in the *SimpleBeanInfo* class to load the icon images:

```
public class ImageViewerBeanBeanInfo extends SimpleBeanInfo
{
   public ImageViewerBeanBeanInfo()
   {
      iconColor16 = loadImage("ImageViewerBean_COLOR_16x16.gif");
      iconColor32 = loadImage("ImageViewerBean_COLOR_32x32.gif");
      iconMono16 = loadImage("ImageViewerBean_MONO_16x16.gif");
      iconMono32 = loadImage("ImageViewerBean_MONO_32x32.gif");
   }

   public Image getIcon(int iconType)
   {
```

```
      if (iconType == BeanInfo.ICON_COLOR_16x16) return iconColor16;
      else if (iconType == BeanInfo.ICON_COLOR_32x32) return
iconColor32;
      else if (iconType == BeanInfo.ICON_MONO_16x16) return
iconMono16;
      else if (iconType == BeanInfo.ICON_MONO_32x32) return
iconMono32;
      else return null;
   }

   private Image iconColor16;
   private Image iconColor32;
   private Image iconMono16;
   private Image iconMono32;
}
```

Property Editors

If you add an integer or string property to a bean, then that property is automatically displayed in the bean's property inspector. But what happens if you add a property whose values cannot easily be edited in a text field, for example, a Date or a Color? Then, you need to provide a separate component by which the user can specify the property value. Such components are called property editors. For example, a property editor for a date object might be a calendar that lets the user scroll through the months and pick a date. A property editor for a Color object would let the user select the red, green, and blue components of the color.

Actually, NetBeans already has a property editor for colors. Also, of course, there are property editors for basic types such as *String (a text field)* and *boolean (a checkbox).*

The process for supplying a new property editor is slightly involved. First, you create a bean info class to accompany your bean. Override the *getPropertyDescriptors* method. That method

returns an array of *PropertyDescriptor* objects. You create one object for each property that should be displayed on a property editor, even those for which you just want the default editor.

You construct a *PropertyDescriptor* by supplying the name of the property and the class of the bean that contains it.

Code View:

PropertyDescriptor descriptor = new PropertyDescriptor("titlePosition", ChartBean.class);

Then you call the *setPropertyEditorClass* method of the *PropertyDescriptor* class.

```
descriptor.setPropertyEditorClass(TitlePositionEditor.class);
```

Next, you build an array of descriptors for properties of your bean. For example, the chart bean that we discuss in this section has five properties:

- A Color property, graphColor
- A String property, title
- An int property, titlePosition
- A double[] property, values
- A boolean property, inverse

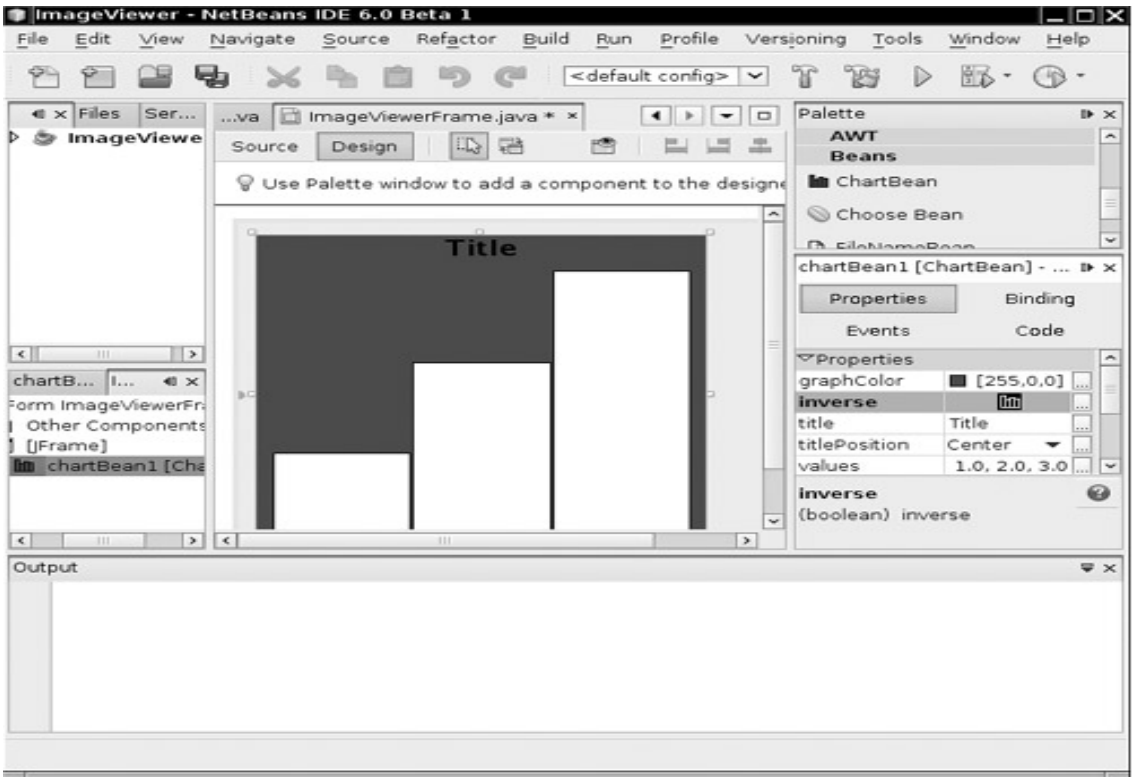
The code in *ChartBeanBeanInfo.java* shows the *ChartBeanBeanInfo* class that specifies the property editors for these properties. It achieves the following:

1. The *getPropertyDescriptors* method returns a descriptor for each property. The title and *graphColor* properties are used with the default editors; that is, the string and color editors that come with the builder tool.
2. The *titlePosition*, values, and inverse properties use special editors of type *TitlePositionEditor, DoubleArrayEditor, and InverseEditor,* respectively.

Figure 30 shows the chart bean. You can see the title on the top. Its position can be set to left, center, or right. The values property specifies the graph values. If the inverse property is true,

then the background is colored and the bars of the chart are white. You can find the code for the chart bean with the book's companion code; the bean is simply a modification of the chart applet in Volume I, Chapter 10.

Figure 30. The chart bean



Writing Property Editors

Before we get into the mechanics of writing property editors, we should point out that a editor is under the control of the builder, not the bean. When the builder displays the property inspector, it carries out the following steps for each bean property.

1. It instantiates a property editor.
2. It asks the bean to tell it the current value of the property.
3. It then asks the property editor to display the value.

A property editor must supply a default constructor, and it must implement the *PropertyEditor interface.* You will usually want to extend the convenience *PropertyEditorSupport* class that provides default versions of these methods.

For every property editor you write, you choose one of three ways to display and edit the property value:

- As a text string (define *getAsText and setAsText*)
- As a choice field (define *getAsText, setAsText, and getTags*)
- Graphically, by painting it (define *isPaintable, paintValue, supportsCustomEditor, and getCustomEditor*)

We have a closer look at these choices in the following sections.

String-Based Property Editors

Simple property editors work with text strings. You override the *setAsText and getAsText* methods. For example, our chart bean has a property that lets you choose where the title should be displayed: Left, Center, or Right. These choices are implemented as an enumeration

public enum Position { LEFT, CENTER, RIGHT };

But of course, we don't want them to appear as uppercase strings LEFT, CENTER, RIGHT—unless we are trying to enter the User Interface Hall of Horrors. Instead, we define a property editor whose *getAsText* method picks a string that looks pleasing to the developer:

```
class TitlePositionEditor extends PropertyEditorSupport
{
   public String getAsText()
   {
      int index = ((ChartBean.Position) getValue()).ordinal();
      return tags[index];
   }
```

```
   . . .
   private String[] tags = { "Left", "Center", "Right" };
}
```

Ideally, these strings should appear in the current locale, not necessarily in English, but we leave that as an exercise to the reader.

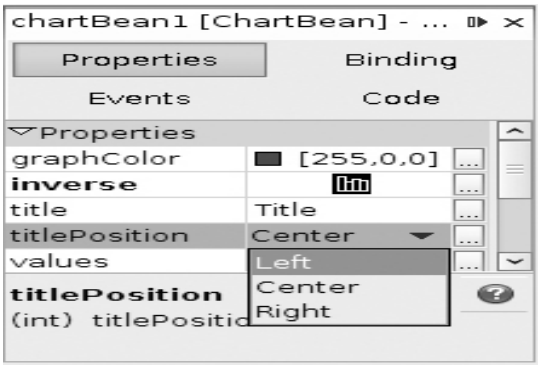
Conversely, we need to supply a method that converts a text string back to the property value:

```
public void setAsText(String s)
{
   int index = Arrays.asList(tags).indexOf(s);
   if (index >= 0) setValue(ChartBean.Position.values()[index]);
}
```

If we simply supply these two methods, the property inspector will provide a text field. It is initialized by a call to *getAsText*, and the *setAsText* method is called when we are done editing. Of course, in our situation, this is not a good choice for the *titlePosition* property, unless, of course, we are also competing for entry into the User Interface Hall of Shame. It is better to display all valid settings in a combo box (see Figure 31). The *PropertyEditorSupport* class gives a simple mechanism for indicating that a combo box is appropriate. Simply write a *getTags* method that returns an array of strings.

```
public String[] getTags() { return tags; }
```

Figure 31. Custom property editors at work



The default *getTags* method returns null, indicating that a text field is appropriate for editing the property value.

When supplying the getTags method, you still need to supply the *getAsText and setAsText* methods. The *getTags* method simply specifies the strings that should be offered to the user. The *getAsText/setAsText* methods translate between the strings and the data type of the property (which can be a string, an integer, an enumeration, or a completely different type).

Finally, property editors should implement the *getJavaInitializationString* method.With this method, you can give the builder tool the Java code that sets a property to its current value. The builder tool uses this string for automatic code generation. Here is the method for the *TitlePositionEditor*:

```
public String getJavaInitializationString()
{
   return ChartBean.Position.class.getName().replace('$', '.') + "." +
getValue();
}
```

This method returns a string such as "*com.horstmann.corejava.ChartBean.Position.LEFT*". Try it out in NetBeans: If you edit the *titlePosition property*, NetBeans inserts code such as

```
chartBean1.setTitlePosition(com.horstmann.corejava.ChartBean.Position.
LEFT);
```

In our situation, the code is a bit cumbersome because *ChartBean.Position.class.getName()* is the string "*com.horstmann.corejava.ChartBean$Position*". We replace the $ with a period, and add the result of invoking *toString* on the enumeration value. If a property has a custom editor that does not implement the getJavaInitializationString method, NetBeans does not know how to generate code and produces a setter with parameter ???.

GUI-Based Property Editors

A sophisticated property should not be edited as text. Instead, a graphical representation is displayed in the property inspector, in the small area that would otherwise hold a text field or combo box. When the user clicks on that area, a custom editor dialog box pops up (see Figure

32). The dialog box contains a component to edit the property values, supplied by the property editor, and various buttons, supplied by the builder environment. In our example, the customizer is rather spare, containing a single button. The book's companion code contains a more elaborate editor for editing the chart values.

Figure 32. A custom editor dialog box

To build a GUI-based property editor, you first tell the property inspector that you will paint the value and not use a string.

Override the *getAsText* method in the *PropertyEditor* interface to return null and the *isPaintable* method to return true.

Then, you implement the *paintValue* method. It receives a Graphics context and the coordinates of the rectangle inside which you can paint. Note that this rectangle is typically small, so you can't have a very elaborate representation. We simply draw one of two icons (which you can see in Figure 31).

```
public void paintValue(Graphics g, Rectangle box)
{
   ImageIcon icon = (Boolean) getValue() ? inverseIcon : normalIcon;
   int x = bounds.x + (bounds.width - icon.getIconWidth()) / 2;
   int y = bounds.y + (bounds.height - icon.getIconHeight()) / 2;
   g.drawImage(icon.getImage(), x, y, null);
```

```
}
```

This graphical representation is not editable. The user must click on it to pop up a custom editor.You indicate that you will have a custom editor by overriding the *supportsCustomEditor* in the *PropertyEditor* interface to return true. Next, the *getCustomEditor* method of the *PropertyEditor* interface constructs and returns an object of the custom editor class. *InverseEditor.java* shows the code for the *InverseEditor* that displays the current property value in the property inspector. *InverseEditorPanel.java* shows the code for the custom editor panel for changing the value.

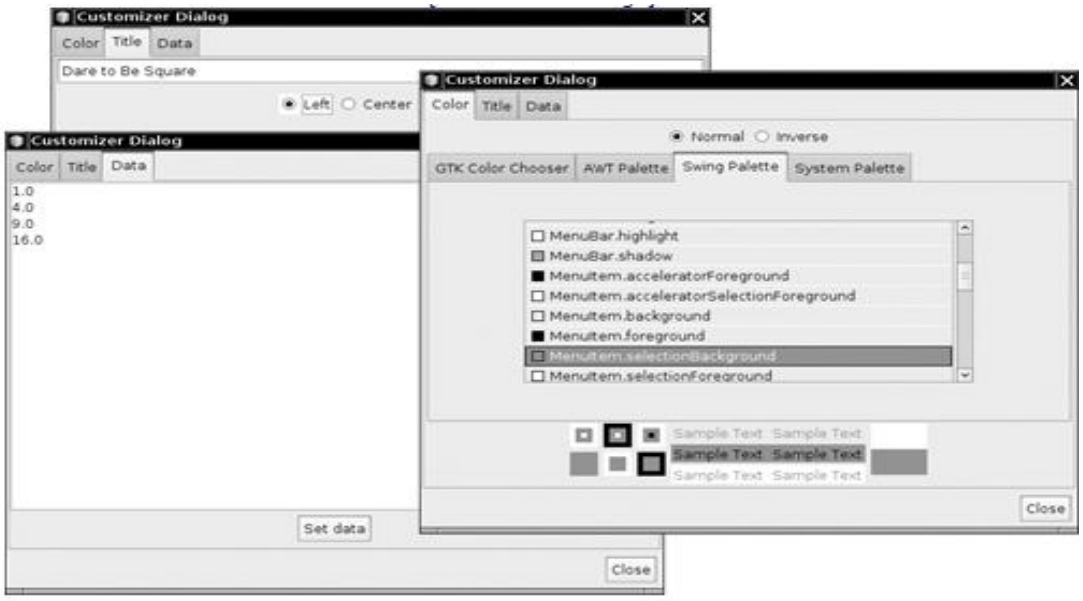
Customizers

A property editor is responsible for allowing the user to set one property at a time. Especially if certain properties of a bean relate to each other, it might be more user friendly to give users a way to edit multiple properties at the same time. To enable this feature, you supply a customizer instead of (or in addition to) multiple property editors.

Moreover, some beans might have features that are not exposed as properties and that therefore cannot be edited through the property inspector. For those beans, a customizer is essential.

In the example program for this section, we develop a customizer for the chart bean. The customizer lets you set several properties of the chart bean in one dialog box, as shown in Figure 33.

Figure 33. The customizer for the ChartBean



To add a customizer to your bean, you must supply a *BeanInfo* class and override the *getBeanDescriptor* method, as shown in the following example.

```
public ChartBean2BeanInfo extends SimpleBeanInfo
{
   public BeanDescriptor getBeanDescriptor()
   {
      return beanDescriptor;
   }
   . . .
   private BeanDescriptor beanDescriptor
      = new BeanDescriptor(ChartBean2.class,
ChartBean2Customizer.class);
}
```

Note that you need not follow any naming pattern for the customizer class. (Nevertheless, it is customary to name the customizer as *BeanNameCustomizer*.)

Writing a Customizer Class

Any customizer class you write must have a default constructor, extend the Component class, and implement the Customizer interface. That interface has only three methods:

- The *setObject* method, which takes a parameter that specifies the bean being customized
- The *addPropertyChangeListener* and *removePropertyChangeListener* methods, which manage the collection of listeners that are notified when a property is changed in the customizer

It is a good idea to update the visual appearance of the target bean by broadcasting a *PropertyChangeEvent* whenever the user changes any of the property values, not just when the user is at the end of the customization process.

Unlike property editors, customizers are not automatically displayed. In *NetBeans*, you must right-click on the bean and select the Customize menu option to pop up the customizer. At that point, the builder calls the *setObject* method of the customizer. Notice that your customizer is created before it is actually linked to an instance of your bean. Therefore, you cannot assume any information about the state of a bean in the constructor.

Because customizers typically present the user with many options, it is often handy to use the tabbed pane user interface. We use this approach and have the customizer extend the *JTabbedPane* class.

The customizer gathers the following information in three panes:

- Graph color and inverse mode
- Title and title position
- Data points

Of course, developing this kind of user interface can be tedious to code—our example devotes over 100 lines just to set it up in the constructor. However, this task requires only the usual Swing programming skills, and we don't dwell on the details here.

One trick is worth keeping in mind. You often need to edit property values in a customizer. Rather than implementing a new interface for setting the property value of a particular class, you can simply locate an existing property editor and add it to your user interface! For example, in our *ChartBean2* customizer, we need to set the graph color. Because we know that *NetBeans* has a perfectly good property editor for colors, we locate it as follows:

```
PropertyEditor colorEditor = PropertyEditorManager.findEditor(Color.Class);
Component colorEditorComponent = colorEditor.getCustomEditor();
```

Once we have all components laid out, we initialize their values in the *setObject* method. The *setObject* method is called when the customizer is displayed. Its parameter is the bean that is being customized. To proceed, we store that bean reference—we'll need it later to notify the bean of property changes. Then, we initialize each user interface component. Here is a part of the *setObject* method of the chart bean customizer that does this initialization:

```
public void setObject(Object obj)
{
   bean = (ChartBean2) obj;
   titleField.setText(bean.getTitle());
   colorEditor.setValue(bean.getGraphColor());
   . . .
}
```

Finally, we hook up event handlers to track the user's activities. Whenever the user changes the value of a component, the component fires an event that our customizer must handle. The event handler must update the value of the property in the bean and must also fire a *PropertyChangeEvent* so that other listeners (such as the property inspector) can be updated. Let us follow that process with a couple of user interface elements in the chart bean customizer.

When the user types a new title, we want to update the title property. We attach a *DocumentListener* to the text field into which the user types the title.

```
titleField.getDocument().addDocumentListener(new
```

```
   DocumentListener()
   {
      public void changedUpdate(DocumentEvent event)
      {
         setTitle(titleField.getText());
      }
      public void insertUpdate(DocumentEvent event)
      {
         setTitle(titleField.getText());
      }
      public void removeUpdate(DocumentEvent event)
      {
         setTitle(titleField.getText());
      }
   });
```

The three listener methods call the setTitle method of the customizer. That method calls the bean to update the property value and then fires a property change event. (This update is necessary only for properties that are not bound.) Here is the code for the setTitle method.

```
public void setTitle(String newValue)
{
   if (bean == null) return;
   String oldValue = bean.getTitle();
   bean.setTitle(newValue);
   firePropertyChange("title", oldValue, newValue);
}
```

When the color value changes in the color property editor, we want to update the graph color of the bean. We track the color changes by attaching a listener to the property editor. Perhaps confusingly, that editor also sends out property change events.

```
colorEditor.addPropertyChangeListener(new
   PropertyChangeListener()
   {
      public void propertyChange(PropertyChangeEvent event)
      {
         setGraphColor((Color) colorEditor.getValue());
      }
   });
```

JavaBeans Persistence

JavaBeans persistence uses JavaBeans properties to save beans to a stream and to read them back at a later time or in a different virtual machine. In this regard, JavaBeans persistence is similar to object serialization. (See Chapter 1 for more information on serialization.) However, there is an important difference: JavaBeans persistence is suitable for long-term storage.

When an object is serialized, its instance fields are written to a stream. If the implementation of a class changes, then its instance fields can change. You cannot simply read files that contain serialized objects of older versions. It is possible to detect version differences and translate between old and new data representations. However, the process is extremely tedious and should only be applied in desperate situations. Plainly, serialization is unsuitable for long-term storage. For that reason, all Swing components have the following message in their documentation: "Warning: Serialized objects of this class will not be compatible with future Swing releases. The current serialization support is appropriate for short term storage or RMI between applications."

The long-term persistence mechanism was invented as a solution for this problem. It was originally intended for drag-and-drop GUI design tools. The design tool saves the result of mouse clicks—a collection of frames, panels, buttons, and other Swing components—in a file,

using the long-term persistence format. The running program simply opens that file. This approach cuts out the tedious source code for laying out and wiring up Swing components. Sadly, it has not been widely implemented.

The basic idea behind JavaBeans persistence is simple. Suppose you want to save a *JFrame* object to a file so that you can retrieve it later. If you look into the source code of the *JFrame* class and its superclasses, then you see dozens of instance fields. If the frame were to be serialized, all of the field values would need to be written. But think about how a frame is constructed:

```
JFrame frame = new JFrame();
frame.setTitle("My Application");
frame.setVisible(true);
```

The default constructor initializes all instance fields, and a couple of properties are set. If you archive the frame object, the JavaBeans persistence mechanism saves exactly these statements in XML format:

```
<object class="javax.swing.JFrame">
   <void property="title">
     <string>My Application</string>
   </void>
   <void property="visible">
     <boolean>true</boolean>
   </void>
</object>
```

When the object is read back, the statements are executed: A *JFrame* object is constructed, and its title and visible properties are set to the given values. It does not matter if the internal representation of the *JFrame* has changed in the meantime. All that matters is that you can restore the object by setting properties.

Note that only those properties that are different from the default are archived. The *XMLEncoder* makes a default *JFrame* and compares its property with the frame that is being archived. Property setter statements are generated only for properties that are different from the default. This process is called redundancy elimination. As a result, the archives are generally smaller than the result of serialization. (When serializing Swing components, the difference is particularly dramatic because Swing objects have a lot of state, most of which is never changed from the default.)

Of course, there are minor technical hurdles with this approach. For example, the call

```
frame.setSize(600, 400);
```

is not a property setter. However, the *XMLEncoder* can cope with this: It writes the statement

```
<void property="bounds">
   <object class="java.awt.Rectangle">
       <int>0</int>
       <int>0</int>
       <int>600</int>
       <int>400</int>
   </object>
</void>
```

To save an object to a stream, use an *XMLEncoder*:

```
XMLEncoder out = new XMLEncoder(new FileOutputStream(. . .));
out.writeObject(frame);
out.close();
```

To read it back, use an *XMLDecoder*:

```
XMLDecoder in = new XMLDecoder(new FileInputStream(. . .));
JFrame newFrame = (JFrame) in.readObject();
in.close();
```

The program in *PersistentFrameTest.java* shows how a frame can load and save itself (see Figure 34). When you run the program, first click the Save button and save the frame to a file. Then move the original frame to a different position and click Load to see another frame pop up at the original location. Have a look inside the XML file that the program produces.

Figure 34. The PersistentFrameTest program

If you look closely at the XML output, you will find that the *XMLEncoder* carries out an amazing amount of work when it saves the frame. The *XMLEncoder* produces statements that carry out the following actions:

- Set various frame properties: size, layout, *defaultCloseOperation*, title, and so on.
- Add buttons to the frame.
- Add action listeners to the buttons.

Here, we had to construct the action listers with the *EventHandler* class. The *XMLEncoder* cannot archive arbitrary inner classes, but it knows how to handle *EventHandler* objects.

Using JavaBeans Persistence for Arbitrary Data

JavaBeans persistence is not limited to the storage of Swing components. You can use the mechanism to store any collection of objects, provided you follow a few simple rules. In the following sections, you learn how you can use JavaBeans persistence as a long-term storage format for your own data.

The *XMLEncoder* has built-in support for the following types:

- null

- All primitive types and their wrappers
- Enumerations (since Java SE 6)
- String
- Arrays
- Collections and maps
- The reflection types Class, Field, Method, and Proxy
- The AWT types Color, Cursor, Dimension, Font, Insets, Point, Rectangle, and ImageIcon
- AWT and Swing components, borders, layout managers, and models
- Event handlers

Writing a Persistence Delegate to Construct an Object

Using JavaBeans persistence is trivial if one can obtain the state of every object by setting properties. But in real programs, there are always a few classes that don't work that way. Consider, for example, the Employee class of Volume I, Chapter 4. Employee isn't a well-behaved bean. It doesn't have a default constructor, and it doesn't have methods *setName*, *setSalary*, *setHireDay*. To overcome this problem, you define a persistence delegate. Such a delegate is responsible for generating an XML encoding of an object.

The persistence delegate for the Employee class overrides the instantiate method to produce an expression that constructs an object.

```
 PersistenceDelegate delegate = new
       DefaultPersistenceDelegate()
       {
          protected Expression instantiate(Object oldInstance, Encoder
out)
          {
```

```
            Employee e = (Employee) oldInstance;
            GregorianCalendar c = new GregorianCalendar();
            c.setTime(e.getHireDay());
            return new Expression(oldInstance, Employee.class, "new",
               new Object[]
               {
                  e.getName(),
                  e.getSalary(),
                  c.get(Calendar.YEAR),
                  c.get(Calendar.MONTH),
                  c.get(Calendar.DATE)
               });
         }
      };
```

This means: "To re-create oldInstance, call the new method (i.e., the constructor) on the *Employee.class* object, and supply the given parameters." The parameter name oldInstance is a bit misleading—this is simply the instance that is being saved.

To install the persistence delegate, you have two choices. You can associate it with a specific *XMLWriter*:

```
out.setPersistenceDelegate(Employee.class, delegate);
```

Alternatively, you can set the *persistenceDelegate* attribute of the bean descriptor of the *BeanInfo:*

```
BeanInfo info = Introspector.getBeanInfo(GregorianCalendar.class);
info.getBeanDescriptor().setValue("persistenceDelegate", delegate);
```

Once the delegate is installed, you can save Employee objects. For example, the statements

```
Object myData = new Employee("Harry Hacker", 50000, 1989, 10, 1);
out.writeObject(myData);
```

generate the following output:

```
<object class="Employee">
   <string>Harry Hacker</string>
   <double>50000.0</double>
   <int>1989</int>
   <int>10</int>
   <int>1</int>
</object>
```

Constructing an Object from Properties

If all constructor parameters can be obtained by accessing properties of *oldInstance*, then you need not write the instantiate method yourself. Instead, simply construct a *DefaultPersistenceDelegate* and supply the property names.

For example, the following statement sets the persistence delegate for the *Rectangle2D.Double* class:

```
out.setPersistenceDelegate(Rectangle2D.Double.class,
 new DefaultPersistenceDelegate(new String[] { "x", "y", "width",
"height" }));
```

This tells the encoder: "To encode a *Rectangle2D.Double* object, get its x, y, width, and height properties and call the constructor with those four values." As a result, the output contains an element such as the following:

```
<object class="java.awt.geom.Rectangle2D$Double">
   <double>5.0</double>
   <double>10.0</double>
   <double>20.0</double>
```

```
   <double>30.0</double>
</object>
```

If you are the author of the class, you can do even better. Annotate the constructor with the *@ConstructorProperties* annotation. Suppose, for example, the Employee class had a constructor with three parameters (name, salary, and hire day). Then we could have annotated the constructor as follows:

```
@ConstructorProperties({"name", "salary", "hireDay"})
public Employee(String n, double s, Date d)
```

This tells the encoder to call the *getName, getSalary, and getHireDay* property getters and write the resulting values into the object expression.

The *@ConstructorProperties* annotation was introduced in Java SE 6, and has so far only been used for classes in the Java Management Extensions (JMX) API.

Constructing an Object with a Factory Method

Sometimes, you need to save objects that are obtained from factory methods, not constructors. Consider, for example, how you get an *InetAddress* object:

```
byte[] bytes = new byte[] { 127, 0, 0, 1};
InetAddress address = InetAddress.getByAddress(bytes);
```

The instantiate method of the PersistenceDelegate produces a call to the factory method.

```
protected Expression instantiate(Object oldInstance, Encoder out)
{
   return new Expression(oldInstance, InetAddress.class,
"getByAddress",
      new Object[] { ((InetAddress) oldInstance).getAddress() });
}
```

A sample output is

```
<object class="java.net.Inet4Address" method="getByAddress">
```

```
   <array class="byte" length="4">
      <void index="0">
         <byte>127</byte>
      </void>
      <void index="3">
         <byte>1</byte>
      </void>
   </array>
</object>
```

Postconstruction Work

The state of some classes is built up by calls to methods that are not property setters. You can cope with that situation by overriding the initialize method of the *DefaultPersistenceDelegate*. The initialize method is called after the instantiate method. You can generate a sequence of statements that are recorded in the archive.

For example, consider the *BitSet* class. To re-create a *BitSet* object, you set all the bits that were present in the original. The following initialize method generates the necessary statements:

```
protected void initialize(Class<?> type, Object oldInstance, Object
newInstance, Encoder out)
{
   super.initialize(type, oldInstance, newInstance, out);
   BitSet bs = (BitSet) oldInstance;
   for (int i = bs.nextSetBit(0); i >= 0; i = bs.nextSetBit(i + 1))
   out.writeStatement(new Statement(bs, "set", new Object[] { i, i +
1, true } ));
}
```

A sample output is

```
<object class="java.util.BitSet">
   <void method="set">
      <int>1</int>
      <int>2</int>
      <boolean>true</boolean>
   </void>
   <void method="set">
      <int>4</int>
      <int>5</int>
      <boolean>true</boolean>
   </void>
</object>
```

Transient Properties

Occasionally, a class has a property with a getter and setter that the *XMLDecoder* discovers, but you don't want to include the property value in the archive. To suppress archiving of a property, mark it as transient in the property descriptor. For example, the following statement marks the *removeMode* property of the *DamageReporter* class (which you will see in detail in the next section) as transient.

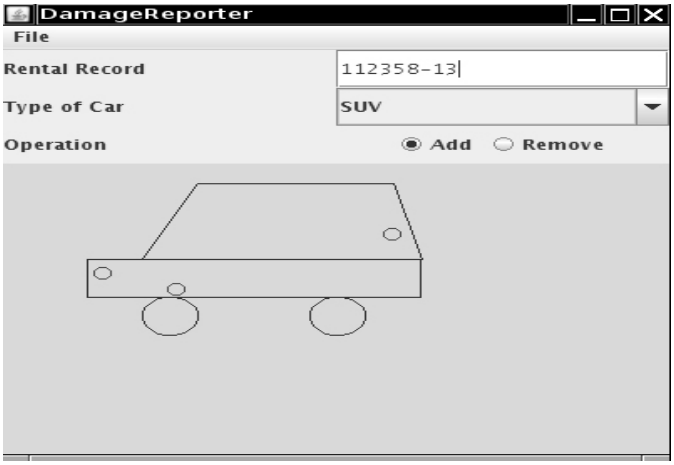
```
BeanInfo info = Introspector.getBeanInfo(DamageReport.class);
for (PropertyDescriptor desc : info.getPropertyDescriptors())
   if (desc.getName().equals("removeMode"))
      desc.setValue("transient", Boolean.TRUE);
```

The program in *PersistentDelegateTest.java* shows the various persistence delegates at work. Keep in mind that this program shows a worst-case scenario—in actual applications, many classes can be archived without the use of delegates.

A Complete Example for JavaBeans Persistence

We end the description of JavaBeans persistence with a complete example (see Figure 35). This application writes a damage report for a rental car. The rental car agent enters the rental record, selects the car type, uses the mouse to click on damaged areas on the car, and saves the report. The application can also load existing damage reports. *DamageReporter.java* contains the code for the program.

Figure 35. The DamageReporter application



The application uses JavaBeans persistence to save and load *DamageReport* objects (see *DamageReport.java*). It illustrates the following aspects of the persistence technology:

- Properties are automatically saved and restored. Nothing needs to be done for the *rentalRecord* and *carType* properties.
- *Postconstruction* work is required to restore the damage locations. The persistence delegate generates statements that call the click method.
- *The Point2D.Double* class needs a *DefaultPersistenceDelegate* that constructs a point from its x and y properties.
- The *removeMode* property (which specifies whether mouse clicks add or remove damage marks) is transient because it should not be saved in damage reports.

Here is a sample damage report:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<java version="1.5.0" class="java.beans.XMLDecoder">
   <object class="DamageReport">
      <object class="java.lang.Enum" method="valueOf">
         <class>DamageReport$CarType</class>
         <string>SEDAN</string>
      </object>
      <void property="rentalRecord">
         <string>12443-19</string>
      </void>
      <void method="click">
         <object class="java.awt.geom.Point2D$Double">
            <double>181.0</double>
            <double>84.0</double>
         </object>
      </void>
      <void method="click">
         <object class="java.awt.geom.Point2D$Double">
            <double>162.0</double>
            <double>66.0</double>
         </object>
      </void>
   </object>
</java>
```

8.2 Demonstration of JavaBeans Components

FileNameBean.java

```
 package com.horstmann.corejava;
  import java.awt.*;
  import java.awt.event.*;
  import java.io.*;
  import java.util.*;
  import javax.swing.*;
  import javax.swing.filechooser.*;
/**
 * A bean for specifying file names.
* @version 1.30 2007-10-03
* @author Cay Horstmann
*/
 public class FileNameBean extends JPanel
 {
public FileNameBean()
     {
       dialogButton = new JButton("...");
       nameField = new JTextField(30);
      chooser = new JFileChooser();
      setPreferredSize(new Dimension(XPREFSIZE, YPREFSIZE));
      setLayout(new GridBagLayout());
     GridBagConstraints gbc = new GridBagConstraints();
     gbc.weightx = 100;
     gbc.weighty = 100;
     gbc.anchor = GridBagConstraints.WEST;
     gbc.fill = GridBagConstraints.BOTH;
     gbc.gridwidth = 1;
     gbc.gridheight = 1;
      add(nameField, gbc);
      dialogButton.addActionListener(new ActionListener()
       {
 public void actionPerformed(ActionEvent event)
           {
          chooser.setFileFilter(new
FileNameExtensionFilter(Arrays.toString(extensions),
                     extensions));
            int r = chooser.showOpenDialog(null);
            if (r == JFileChooser.APPROVE_OPTION)
               {
                 File f = chooser.getSelectedFile();
                 String name = f.getAbsolutePath();
```

```
                setFileName(name);
              }
             }
          });
        nameField.setEditable(false);
        gbc.weightx = 0;
        gbc.anchor = GridBagConstraints.EAST;
        gbc.fill = GridBagConstraints.NONE;
        gbc.gridx = 1;
        add(dialogButton, gbc);
    }
     /**
     * Sets the fileName property.
      * @param newValue the new file name
      */
public void setFileName(String newValue)
     {
       String oldValue = nameField.getText();
      nameField.setText(newValue);
        firePropertyChange("fileName", oldValue, newValue);
     }

    /**
 * Gets the fileName property.
 * @return the name of the selected file
*/
public String getFileName()
    {
      return nameField.getText();
     }

 /**
* Gets the extensions property.
 * @return the default extensions in the file chooser
 */
public String[] getExtensions()
     {
        return extensions;
    }

 /**
* Sets the extensions property.
* @param newValue the new default extensions
*/
public void setExtensions(String[] newValue)
```

```
{
     extensions = newValue;
 }
/**
 * Gets one of the extensions property values.
* @param i the index of the property value
* @return the value at the given index
*/
public String getExtensions(int i)
{
        if (0 <= i && i < extensions.length) return extensions[i];
      else return "";
    }
 /**
* Sets one of the extensions property values.
* @param i the index of the property value
* @param newValue the new value at the given index
*/
public void setExtensions(int i, String newValue)
{
       if (0 <= i && i < extensions.length) extensions[i] = newValue;
    }
    private static final int XPREFSIZE = 200;
   private static final int YPREFSIZE = 20;
   private JButton dialogButton;
   private JTextField nameField;
   private JFileChooser chooser;
   private String[] extensions = { "gif", "png" };
}
```

java.beans.PropertyChangeListener 1.1

- void propertyChange(PropertyChangeEvent event)

is called when a property change event is fired.

java.beans.PropertyChangeSupport 1.1

- PropertyChangeSupport(Object sourceBean)

constructs a PropertyChangeSupport object that manages listeners for bound property changes of the given bean.

- void addPropertyChangeListener(PropertyChangeListener listener)
- void addPropertyChangeListener(String propertyName, PropertyChangeListener listener) 1.2

registers an interested listener for changes in all bound properties, or only the named bound property.

- void removePropertyChangeListener(PropertyChangeListener listener)
- void removePropertyChangeListener(String propertyName, PropertyChangeListener listener) 1.2

removes a previously registered property change listener.

- void firePropertyChange(String propertyName, Object oldValue, Object newValue)
- void firePropertyChange(String propertyName, int oldValue, int newValue) 1.2
- void firePropertyChange(String propertyName, boolean oldValue, boolean newValue) 1.2

sends a PropertyChangeEvent to registered listeners.

- void fireIndexedPropertyChange(String propertyName, int index, Object oldValue, Object newValue) 5.0
- void fireIndexedPropertyChange(String propertyName, int index, int oldValue, int newValue) 5.0
- void fireIndexedPropertyChange(String propertyName, int index, boolean oldValue, boolean newValue) 5.0

sends an IndexedPropertyChangeEvent to registered listeners.

- PropertyChangeListener[] getPropertyChangeListeners() 1.4
- PropertyChangeListener[] getPropertyChangeListeners(String propertyName) 1.4

gets the listeners for changes in all bound properties, or only the named bound property.

java.beans.PropertyChangeEvent 1.1

- PropertyChangeEvent(Object sourceBean, String propertyName, Object oldValue, Object newValue)

constructs a new PropertyChangeEvent object, describing that the given property has changed from oldValue to newValue.

- String getPropertyName()

returns the name of the property.

- Object getOldValue();
- Object getNewValue()

returns the old and new value of the property.

java.beans.IndexedPropertyChangeEvent 5.0

- IndexedPropertyChangeEvent(Object sourceBean, String propertyName, int index, Object oldValue, Object newValue)

constructs a new IndexedPropertyChangeEvent object, describing that the given property has changed from oldValue to newValue at the given index.

- int getIndex()

returns the index at which the change occurred.

java.beans.VetoableChangeListener 1.1

- void vetoableChange(PropertyChangeEvent event)

is called when a property is about to be changed. It should throw a PropertyVetoException if the change is not acceptable.

java.beans.VetoableChangeSupport 1.1

- VetoableChangeSupport(Object sourceBean)

constructs a PropertyChangeSupport object that manages listeners for constrained property changes of the given bean.

- void addVetoableChangeListener(VetoableChangeListener listener)
- void addVetoableChangeListener(String propertyName, VetoableChangeListener listener) 1.2

registers an interested listener for changes in all constrained properties, or only the named constrained property.

- void removeVetoableChangeListener(VetoableChangeListener listener)
- void removeVetoableChangeListener(String propertyName, VetoableChangeListener listener) 1.2

removes a previously registered vetoable change listener.

- void fireVetoableChange(String propertyName, Object oldValue, Object newValue)
- void fireVetoableChange(String propertyName, int oldValue, int newValue) 1.2
- void fireVetoableChange(String propertyName, boolean oldValue, boolean newValue) 1.2

sends a VetoableChangeEvent to registered listeners.

- VetoableChangeListener[] getVetoableChangeListeners() 1.4
- VetoableChangeListener[] getVetoableChangeListeners(String propertyName) 1.4

gets the listeners for changes in all constrained properties, or only the named bound property.

java.awt.Component 1.0

- void addPropertyChangeListener(PropertyChangeListener listener) 1.2
- void addPropertyChangeListener(String propertyName, PropertyChangeListener listener) 1.2

registers an interested listener for changes in all bound properties, or only the named bound property.

- void removePropertyChangeListener(PropertyChangeListener listener) 1.2
- void removePropertyChangeListener(String propertyName, PropertyChangeListener listener) 1.2

removes a previously registered property change listener.

- void firePropertyChange(String propertyName, Object oldValue, Object newValue) 1.2

sends a PropertyChangeEvent to registered listeners.

javax.swing.JComponent 1.2

- void addVetoableChangeListener(VetoableChangeListener listener)

registers an interested listener for changes in all constrained properties, or only the named

constrained property.

- void removeVetoableChangeListener(VetoableChangeListener listener)

removes a previously registered vetoable change listener.

- void fireVetoableChange(String propertyName, Object oldValue, Object newValue)

sends a VetoableChangeEvent to registered listeners.

java.beans.PropertyVetoException 1.1

- PropertyVetoException(String message, PropertyChangeEvent event)

creates a new PropertyVetoException.

- PropertyChangeEvent getPropertyChangeEvent()

returns the PropertyChangeEvent that was vetoed.

java.beans.Introspector 1.1

- static BeanInfo getBeanInfo(Class<?> beanClass)

gets the bean information of the given class.

java.beans.BeanInfo 1.1

- PropertyDescriptor[] getPropertyDescriptors()

returns the descriptors for the bean properties. A return of null indicates that the naming conventions should be used to find the properties.

- Image getIcon(int iconType)

returns an image object that can represent the bean in toolboxes, tool bars, and the like. There are four constants, as described earlier, for the standard types of icons.

java.beans.SimpleBeanInfo 1.1

- Image loadImage(String resourceName)

returns an image object file associated to the resource. The resource name is a path name, taken relative to the directory containing the bean info class.

java.beans.FeatureDescriptor 1.1

- String getName()
- void setName(String name)

gets or sets the programmatic name for the feature.

- String getDisplayName()
- void setDisplayName(String displayName)

gets or sets a display name for the feature. The default value is the value returned by getName. However, currently there is no explicit support for supplying feature names in multiple locales.

- String getShortDescription()
- void setShortDescription(String text)

gets or sets a string that a builder tool can use to provide a short description for this feature. The default value is the return value of getDisplayName.

- boolean isExpert()
- void setExpert(boolean b)

gets or sets an expert flag that a builder tool can use to determine whether to hide the feature from a naive user.

- boolean isHidden()
- void setHidden(boolean b)

gets or sets a flag that a builder tool should hide this feature.

java.beans.PropertyDescriptor 1.1

- PropertyDescriptor(String propertyName, Class<?> beanClass)
- PropertyDescriptor(String propertyName, Class<?> beanClass, String getMethod, String setMethod)

constructs a PropertyDescriptor object. The methods throw an IntrospectionException if an error occurred during introspection. The first constructor assumes that you follow the standard convention for the names of the get and set methods.

- Class<?> getPropertyType()

returns a Class object for the property type.

- Method getReadMethod()
- Method getWriteMethod()

returns the method to get or set the property.

java.beans.IndexedPropertyDescriptor 1.1

- IndexedPropertyDescriptor(String propertyName, Class<?> beanClass)
- IndexedPropertyDescriptor(String propertyName, Class<?> beanClass, String getMethod, String setMethod, String indexedGetMethod, String indexedSetMethod)

constructs an IndexedPropertyDescriptor for the index property. The first constructor assumes that you follow the standard convention for the names of the get and set methods.

- Method getIndexedReadMethod()
- Method getIndexedWriteMethod()

returns the method to get or set an indexed value in the property.

ChartBeanBeanInfo.java

```
package com.horstmann.corejava;
import java.awt.*;
import java.beans.*;
/**
```

```
* The bean info for the chart bean, specifying the property editors.
* @version 1.20 2007-10-05
* @author Cay Horstmann
*/
public class ChartBeanBeanInfo extends SimpleBeanInfo
{
public ChartBeanBeanInfo()
{
       iconColor16 = loadImage("ChartBean_COLOR_16x16.gif");
       iconColor32 = loadImage("ChartBean_COLOR_32x32.gif");
       iconMono16 = loadImage("ChartBean_MONO_16x16.gif");
      iconMono32 = loadImage("ChartBean_MONO_32x32.gif");
       try
       {
          PropertyDescriptor titlePositionDescriptor = new
PropertyDescriptor("titlePosition",
                ChartBean.class);

titlePositionDescriptor.setPropertyEditorClass(TitlePositionEditor.cl
ass);
          PropertyDescriptor inverseDescriptor = new
PropertyDescriptor("inverse", ChartBean.class);

inverseDescriptor.setPropertyEditorClass(InverseEditor.class);
          PropertyDescriptor valuesDescriptor = new
PropertyDescriptor("values", ChartBean.class);

valuesDescriptor.setPropertyEditorClass(DoubleArrayEditor.class);
          propertyDescriptors = new PropertyDescriptor[] {
                new PropertyDescriptor("title", ChartBean.class),
titlePositionDescriptor,
                valuesDescriptor, new
PropertyDescriptor("graphColor", ChartBean.class),
                inverseDescriptor };
        }
        catch (IntrospectionException e)
        {
           e.printStackTrace();
        }
    }

public PropertyDescriptor[] getPropertyDescriptors()
    {
       return propertyDescriptors;
}
```

```
public Image getIcon(int iconType)
    {
       if (iconType == BeanInfo.ICON_COLOR_16x16) return iconColor16;
       else if (iconType == BeanInfo.ICON_COLOR_32x32) return
iconColor32;
       else if (iconType == BeanInfo.ICON_MONO_16x16) return
iconMono16;
       else if (iconType == BeanInfo.ICON_MONO_32x32) return
iconMono32;
       else return null;
 }
    private PropertyDescriptor[] propertyDescriptors;
    private Image iconColor16;
    private Image iconColor32;
    private Image iconMono16;
    private Image iconMono32;
}
```

java.beans.PropertyDescriptor 1.1

- PropertyDescriptor(String name, Class<?> beanClass)

constructs a PropertyDescriptor object.

Parameters:	name	The name of the property
	beanClass	The class of the bean to which the property belongs

- void setPropertyEditorClass(Class<?> editorClass)

sets the class of the property editor to be used with this property.

java.beans.BeanInfo 1.1

- PropertyDescriptor[] getPropertyDescriptors()

returns a descriptor for each property that should be displayed in the property inspector for the bean.

PersistentFrameTest.java

```
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.io.*;
import javax.swing.*;
/**
```

```
* This program demonstrates the use of an XML encoder and decoder to
save and restore a frame.
* @version 1.01 2007-10-03
* @author Cay Horstmann
*/
public class PersistentFrameTest
{
 public static void main(String[] args)
    {
       chooser = new JFileChooser();
       chooser.setCurrentDirectory(new File("."));
       PersistentFrameTest test = new PersistentFrameTest();
       test.init();
    }
    public void init()
    {
       frame = new JFrame();
       frame.setLayout(new FlowLayout());
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       frame.setTitle("PersistentFrameTest");
       frame.setSize(400, 200);
       JButton loadButton = new JButton("Load");
       frame.add(loadButton);

loadButton.addActionListener(EventHandler.create(ActionListener.class
, this, "load"));
       JButton saveButton = new JButton("Save");
       frame.add(saveButton);

saveButton.addActionListener(EventHandler.create(ActionListener.class
, this, "save"));
       frame.setVisible(true);
    }
public void load()
    {
 // show file chooser dialog
       int r = chooser.showOpenDialog(null);
// if file selected, open
       if(r == JFileChooser.APPROVE_OPTION)
       {
          try
          {
             File file = chooser.getSelectedFile();
             XMLDecoder decoder = new XMLDecoder(new
FileInputStream(file));
```

```
            decoder.readObject();
            decoder.close();
         }
         catch (IOException e)
         {
            JOptionPane.showMessageDialog(null, e);
         }
   }
    }
public void save()
    {
       if (chooser.showSaveDialog(null) ==
JFileChooser.APPROVE_OPTION)
       {
          try
         {
             File file = chooser.getSelectedFile();
            XMLEncoder encoder = new XMLEncoder(new
FileOutputStream(file));
            encoder.writeObject(frame);
             encoder.close();
         }
         catch (IOException e)
         {
            JOptionPane.showMessageDialog(null, e);
         }
    }
 }
    private static JFileChooser chooser;
    private JFrame frame;
}
```

TitlePositionEditor.java

```
 package com.horstmann.corejava;
import java.beans.*;
import java.util.*;
 /**
 * A custom editor for the titlePosition property of the ChartBean.
The editor lets the user
 * choose between Left, Center, and Right
 * @version 1.20 2007-12-14
* @author Cay Horstmann
*/
public class TitlePositionEditor extends PropertyEditorSupport
{
public String[] getTags()
{
       return tags;
   }
public String getJavaInitializationString()
{
      return ChartBean.Position.class.getName().replace('$', '.') +
"." + getValue();
}
public String getAsText()
{
    int index = ((ChartBean.Position) getValue()).ordinal();
    return tags[index];
 }
public void setAsText(String s)
  {
       int index = Arrays.asList(tags).indexOf(s);
       if (index >= 0) setValue(ChartBean.Position.values()[index]);
   }

   private String[] tags = { "Left", "Center", "Right" };
}
```

InverseEditor.java

```
 package com.horstmann.corejava;

 import java.awt.*;
 import java.beans.*;
 import javax.swing.*;
 /**
  * The property editor for the inverse property of the ChartBean.
The inverse property toggles
  * between colored graph bars and colored background.
  * @version 1.30 2007-10-03
  * @author Cay Horstmann
  */
public class InverseEditor extends PropertyEditorSupport
{
public Component getCustomEditor()
{
       return new InverseEditorPanel(this);
    }
public boolean supportsCustomEditor()
{
       return true;
    }
public boolean isPaintable()
    {
      return true;
   }
public String getAsText()
    {
       return null;
    }
public String getJavaInitializationString()
    {
       return "" + getValue();
    }
public void paintValue(Graphics g, Rectangle bounds)
{
     ImageIcon icon = (Boolean) getValue() ? inverseIcon :
normalIcon;
     int x = bounds.x + (bounds.width - icon.getIconWidth()) / 2;
    int y = bounds.y + (bounds.height - icon.getIconHeight()) / 2;
      g.drawImage(icon.getImage(), x, y, null);
    }
 private ImageIcon inverseIcon = new
ImageIcon(getClass().getResource(
```

```
          "ChartBean_INVERSE_16x16.gif"));
    private ImageIcon normalIcon =
         new
ImageIcon(getClass().getResource("ChartBean_MONO_16x16.gif"));
 }
```

InverseEditorPanel.java

```
package com.horstmann.corejava;
import java.awt.event.*;
import java.beans.*;
import javax.swing.*;
/**
* The panel for setting the inverse property. It contains a button to
toggle between normal
* and inverse coloring.
* @version 1.30 2007-10-03
* @author Cay Horstmann
*/
public class InverseEditorPanel extends JPanel
{
public InverseEditorPanel(PropertyEditorSupport ed)
   {
      editor = ed;
      button = new JButton();
       updateButton();
      button.addActionListener(new ActionListener()
          {
public void actionPerformed(ActionEvent event)
            {
                editor.setValue(!(Boolean) editor.getValue());
               updateButton();
             }
         });
       add(button);
   }
 private void updateButton()
    {
      if ((Boolean) editor.getValue())
       {
         button.setIcon(inverseIcon);
          button.setText("Inverse");
      }
```

```
        else
       {
          button.setIcon(normalIcon);
          button.setText("Normal");
        }
    }
    private JButton button;
    private PropertyEditorSupport editor;
    private ImageIcon inverseIcon = new
ImageIcon(getClass().getResource(
          "ChartBean_INVERSE_16x16.gif"));
   private ImageIcon normalIcon =
    new
ImageIcon(getClass().getResource("ChartBean_MONO_16x16.gif"));
}
```

java.beans.PropertyEditor 1.1

- Object getValue()

returns the current value of the property. Basic types are wrapped into object wrappers.

- void setValue(Object newValue)

sets the property to a new value. Basic types must be wrapped into object wrappers.

Parameters:	newValue	The new value of the object; should be a newly created object that the property can own

- String getAsText()

override this method to return a string representation of the current value of the property. The default returns null to indicate that the property cannot be represented as a string.

- void setAsText(String text)

override this method to set the property to a new value that is obtained by parsing the text. May throw an IllegalArgumentException if the text does not represent a legal value or if this property cannot be represented as a string.

- String[] getTags()

override this method to return an array of all possible string representations of the property values so they can be displayed in a Choice box. The default returns null to indicate that there is not a finite set of string values.

- boolean isPaintable()

override this method to return true if the class uses the paintValue method to display the property.

- void paintValue(Graphics g, Rectangle bounds)

override this method to represent the value by drawing into a graphics context in the specified place on the component used for the property inspector.

- boolean supportsCustomEditor()

override this method to return true if the property editor has a custom editor.

- Component getCustomEditor()

override this method to return the component that contains a customized GUI for editing the property value.

- String getJavaInitializationString()

override this method to return a Java code string that can be used to generate code that initializes the property value. Examples are "0", "new Color(64, 64, 64)".

java.beans.BeanInfo 1.1

- BeanDescriptor getBeanDescriptor()

returns a BeanDescriptor object that describes features of the bean.

java.beans.BeanDescriptor 1.1

- BeanDescriptor(Class<?> beanClass, Class<?> customizerClass)

constructs a BeanDescriptor object for a bean that has a customizer.

Parameters:	beanClass	The Class object for the bean
	customizerClass	The Class object for the bean's customizer

ChartBean2Customizer.java

```
 package com.horstmann.corejava;

  import java.awt.*;
  import java.awt.event.*;
  import java.beans.*;
  import java.util.*;
  import javax.swing.*;
  import javax.swing.event.*;

 /**
 * A customizer for the chart bean that allows the user to edit all
chart properties in a
 * single tabbed dialog.
 * @version 1.12 2007-10-03
 * @author Cay Horstmann
 */
 public class ChartBean2Customizer extends JTabbedPane implements
Customizer
 {
 public ChartBean2Customizer()
 {
       data = new JTextArea();
```

```
       JPanel dataPane = new JPanel();
       dataPane.setLayout(new BorderLayout());
      dataPane.add(new JScrollPane(data), BorderLayout.CENTER);
      JButton dataButton = new JButton("Set data");
      dataButton.addActionListener(new ActionListener()
          {
 public void actionPerformed(ActionEvent event)
           {
               setData(data.getText());
           }
          });
       JPanel panel = new JPanel();
       panel.add(dataButton);
       dataPane.add(panel, BorderLayout.SOUTH);
       JPanel colorPane = new JPanel();
      colorPane.setLayout(new BorderLayout());
       normal = new JRadioButton("Normal", true);
      inverse = new JRadioButton("Inverse", false);
      panel = new JPanel();
      panel.add(normal);
      panel.add(inverse);
      ButtonGroup group = new ButtonGroup();
      group.add(normal);
      group.add(inverse);
      normal.addActionListener(new ActionListener()
         {
 public void actionPerformed(ActionEvent event)
           {
               setInverse(false);
            }
         });
        inverse.addActionListener(new ActionListener()
          {
 public void actionPerformed(ActionEvent event)
             {
               setInverse(true);
            }
          });
      colorEditor = PropertyEditorManager.findEditor(Color.class);
      colorEditor.addPropertyChangeListener(new
PropertyChangeListener()
           {
 public void propertyChange(PropertyChangeEvent event)
             {
               setGraphColor((Color) colorEditor.getValue());
```

```
            }
          });
       colorPane.add(panel, BorderLayout.NORTH);
       colorPane.add(colorEditor.getCustomEditor(),
BorderLayout.CENTER);
       JPanel titlePane = new JPanel();
       titlePane.setLayout(new BorderLayout());
      group = new ButtonGroup();
      position = new JRadioButton[3];
      position[0] = new JRadioButton("Left");
      position[1] = new JRadioButton("Center");
      position[2] = new JRadioButton("Right");
      panel = new JPanel();
      for (int i = 0; i < position.length; i++)
     {
    final ChartBean2.Position pos = ChartBean2.Position.values()[i];
          panel.add(position[i]);
          group.add(position[i]);
         position[i].addActionListener(new ActionListener()
            {
 public void actionPerformed(ActionEvent event)
               {
                  setTitlePosition(pos);
                }
            });
        }
         titleField = new JTextField();
         titleField.getDocument().addDocumentListener(new
DocumentListener()
          {
public void changedUpdate(DocumentEvent evt)
             {
                setTitle(titleField.getText());
             }
public void insertUpdate(DocumentEvent evt)
             {
                setTitle(titleField.getText());
             }
public void removeUpdate(DocumentEvent evt)
             {
                setTitle(titleField.getText());
             }
          });
       titlePane.add(titleField, BorderLayout.NORTH);
       JPanel panel2 = new JPanel();
```

```
      panel2.add(panel);
       titlePane.add(panel2, BorderLayout.CENTER);
       addTab("Color", colorPane);
       addTab("Title", titlePane);
       addTab("Data", dataPane);
   }
/**
* Sets the data to be shown in the chart.
* @param s a string containing the numbers to be displayed, separated
by white space
*/
public void setData(String s)
   {
      StringTokenizer tokenizer = new StringTokenizer(s);
     int i = 0;
     double[] values = new double[tokenizer.countTokens()];
     while (tokenizer.hasMoreTokens())
     {
        String token = tokenizer.nextToken();
        try
         {
             values[i] = Double.parseDouble(token);
            i++;
          }
         catch (NumberFormatException e)
         {
          }
     }
       setValues(values);
  }
/**
* Sets the title of the chart.
* @param newValue the new title
*/
public void setTitle(String newValue)
    {
       if (bean == null) return;
       String oldValue = bean.getTitle();
       bean.setTitle(newValue);
       firePropertyChange("title", oldValue, newValue);
    }
 /**
  * Sets the title position of the chart.
  * @param i the new title position (ChartBean2.LEFT,
ChartBean2.CENTER, or ChartBean2.RIGHT)
```

```
*/
public void setTitlePosition(ChartBean2.Position pos)
    {
       if (bean == null) return;
       ChartBean2.Position oldValue = bean.getTitlePosition();
       bean.setTitlePosition(pos);
       firePropertyChange("titlePosition", oldValue, pos);
}
 /**
* Sets the inverse setting of the chart.
* @param b true if graph and background color are inverted
*/
public void setInverse(boolean b)
{
      if (bean == null) return;
      boolean oldValue = bean.isInverse();
      bean.setInverse(b);
      firePropertyChange("inverse", oldValue, b);
  }
/**
* Sets the values to be shown in the chart.
* @param newValue the new value array
*/
public void setValues(double[] newValue)
    {
       if (bean == null) return;
       double[] oldValue = bean.getValues();
      bean.setValues(newValue);
       firePropertyChange("values", oldValue, newValue);
  }
 /**
* Sets the color of the chart
* @param newValue the new color
*/
public void setGraphColor(Color newValue)
{
       if (bean == null) return;
       Color oldValue = bean.getGraphColor();
       bean.setGraphColor(newValue);
       firePropertyChange("graphColor", oldValue, newValue);
    }
public void setObject(Object obj)
{
      bean = (ChartBean2) obj;
       data.setText("");
```

```
        for (double value : bean.getValues())
         data.append(value + "\n");
        normal.setSelected(!bean.isInverse());
        inverse.setSelected(bean.isInverse());
        titleField.setText(bean.getTitle());
        for (int i = 0; i < position.length; i++)
          position[i].setSelected(i ==
bean.getTitlePosition().ordinal());
          colorEditor.setValue(bean.getGraphColor());
}
    private ChartBean2 bean;
    private PropertyEditor colorEditor;
    private JTextArea data;
    private JRadioButton normal;
    private JRadioButton inverse;
    private JRadioButton[] position;
    private JTextField titleField;
}
```

java.beans.Customizer 1.1

- void setObject(Object bean)

specifies the bean to customize.

PersistenceDelegateTest.java

```
import java.awt.geom.*;
import java.beans.*;
import java.net.*;
import java.util.*;
/**
* This program demonstrates various persistence delegates.
* @version 1.01 2007-10-03
* @author Cay Horstmann
*/
public class PersistenceDelegateTest
 {
public static class Point
    {
       @ConstructorProperties( { "x", "y" })
public Point(int x, int y)
```

```
       {
          this.x = x;
          this.y = y;
}
public int getX()
       {
          return x;
            }
 public int getY()
       {
         return y;
       }
       private final int x, y;
    }
public static void main(String[] args) throws Exception
    {
       PersistenceDelegate delegate = new PersistenceDelegate()
          {
             protected Expression instantiate(Object oldInstance,
Encoder out)
             {
                Employee e = (Employee) oldInstance;
                GregorianCalendar c = new GregorianCalendar();
                c.setTime(e.getHireDay());
                return new Expression(oldInstance, Employee.class,
"new", new Object[] {
                      e.getName(), e.getSalary(),
c.get(Calendar.YEAR), c.get(Calendar.MONTH),
                      c.get(Calendar.DATE) });
             }
          };
      BeanInfo info = Introspector.getBeanInfo(Employee.class);
      info.getBeanDescriptor().setValue("persistenceDelegate",
delegate);
       XMLEncoder out = new XMLEncoder(System.out);
       out.setExceptionListener(new ExceptionListener()
          {
public void exceptionThrown(Exception e)
          {
                e.printStackTrace();
            }
          });
out.setPersistenceDelegate(Rectangle2D.Double.class, new
DefaultPersistenceDelegate(
             new String[] { "x", "y", "width", "height" }));
```

```
       out.setPersistenceDelegate(Inet4Address.class, new
DefaultPersistenceDelegate()
          {
             protected Expression instantiate(Object oldInstance,
Encoder out)
             {
                return new Expression(oldInstance, InetAddress.class,
"getByAddress",
                      new Object[] { ((InetAddress)
oldInstance).getAddress() });
             }
          });
       out.setPersistenceDelegate(BitSet.class, new
DefaultPersistenceDelegate()
          {
             protected void initialize(Class<?> type, Object
oldInstance, Object newInstance,
                   Encoder out)
             {
                super.initialize(type, oldInstance, newInstance,
out);
                BitSet bs = (BitSet) oldInstance;
                for (int i = bs.nextSetBit(0); i >= 0; i =
bs.nextSetBit(i + 1))
                   out.writeStatement(new Statement(bs, "set",
                                                    new Object[] { i,
i + 1, true }));
             }
          });
       out.writeObject(new Employee("Harry Hacker", 50000, 1989, 10,
1));
       out.writeObject(new Point(17, 29));
       out.writeObject(new java.awt.geom.Rectangle2D.Double(5, 10,
20, 30));
       out.writeObject(InetAddress.getLocalHost());
       BitSet bs = new BitSet();
      bs.set(1, 4);
      bs.clear(2, 3);
      out.writeObject(bs);
      out.close();
    }
}
```

DamageReporter.java

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.beans.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
/**
* This program demonstrates the use of an XML encoder and decoder.
All GUI and drawing
* code is collected in this class. The only interesting pieces are
the action listeners for
* openItem and saveItem. Look inside the DamageReport class for
encoder customizations.
* @version 1.01 2004-10-03
* @author Cay Horstmann
*/
 public class DamageReporter extends JFrame
 {
 public static void main(String[] args)
     {
        JFrame frame = new DamageReporter();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
public DamageReporter()
     {
        setTitle("DamageReporter");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
         chooser = new JFileChooser();
        chooser.setCurrentDirectory(new File("."));
            report = new DamageReport();
        report.setCarType(DamageReport.CarType.SEDAN);
        // set up the menu bar
        JMenuBar menuBar = new JMenuBar();
        setJMenuBar(menuBar);
        JMenu menu = new JMenu("File");
        menuBar.add(menu);
       JMenuItem openItem = new JMenuItem("Open");
       menu.add(openItem);
       openItem.addActionListener(new ActionListener()
           {
 public void actionPerformed(ActionEvent evt)
             {
               // show file chooser dialog
```

```
                int r = chooser.showOpenDialog(null);
               // if file selected, open
                if (r == JFileChooser.APPROVE_OPTION)
               {
                   try
                    {
                    File file = chooser.getSelectedFile();
                    XMLDecoder decoder = new XMLDecoder(new
FileInputStream(file));
                     report = (DamageReport) decoder.readObject();
                    decoder.close();
                    rentalRecord.setText(report.getRentalRecord());
                    carType.setSelectedItem(report.getCarType());
                    repaint();
                   }
                    catch (IOException e)
                    {
                       JOptionPane.showMessageDialog(null, e);
                    }
              }
             }
           });
       JMenuItem saveItem = new JMenuItem("Save");
       menu.add(saveItem);
       saveItem.addActionListener(new ActionListener()
           {
public void actionPerformed(ActionEvent evt)
             {
              report.setRentalRecord(rentalRecord.getText());
           chooser.setSelectedFile(new File(rentalRecord.getText() +
".xml"));

            // show file chooser dialog
            int r = chooser.showSaveDialog(null);
             // if file selected, save
             if (r == JFileChooser.APPROVE_OPTION)
                 {
                   try
                   {
                       File file = chooser.getSelectedFile();
                       XMLEncoder encoder = new XMLEncoder(new
FileOutputStream(file));
                       report.configureEncoder(encoder);
                       encoder.writeObject(report);
                       encoder.close();
```

```
                    }
                    catch (IOException e)
                    {
                       JOptionPane.showMessageDialog(null, e);
                    }
                 }
              }
           });
        JMenuItem exitItem = new JMenuItem("Exit");
        menu.add(exitItem);
       exitItem.addActionListener(new ActionListener()
           {
public void actionPerformed(ActionEvent event)
              {
                 System.exit(0);
              }
           });
// combo box for car type
        rentalRecord = new JTextField();
        carType = new JComboBox();
        carType.addItem(DamageReport.CarType.SEDAN);
        carType.addItem(DamageReport.CarType.WAGON);
        carType.addItem(DamageReport.CarType.SUV);
        carType.addActionListener(new ActionListener()
           {
public void actionPerformed(ActionEvent event)
              {
                 DamageReport.CarType item = (DamageReport.CarType)
carType.getSelectedItem();
                 report.setCarType(item);
                 repaint();
              }
           });
// component for showing car shape and damage locations
        carComponent = new JComponent()
           {
public void paintComponent(Graphics g)
              {
                 Graphics2D g2 = (Graphics2D) g;
                 g2.setColor(new Color(0.9f, 0.9f, 0.45f));
                 g2.fillRect(0, 0, getWidth(), getHeight());
                 g2.setColor(Color.BLACK);
                 g2.draw(shapes.get(report.getCarType()));
                 report.drawDamage(g2);
              }
```

```
          };
       carComponent.addMouseListener(new MouseAdapter()
      {
public void mousePressed(MouseEvent event)
           {
               report.click(new Point2D.Double(event.getX(),
event.getY()));
               repaint();
            }
          });
       // radio buttons for click action
       addButton = new JRadioButton("Add");
       removeButton = new JRadioButton("Remove");
       ButtonGroup group = new ButtonGroup();
       JPanel buttonPanel = new JPanel();
       group.add(addButton);
       buttonPanel.add(addButton);
       group.add(removeButton);
       buttonPanel.add(removeButton);
       addButton.setSelected(!report.getRemoveMode());
       removeButton.setSelected(report.getRemoveMode());
       addButton.addActionListener(new ActionListener()
          {
             public void actionPerformed(ActionEvent event)
             {
                report.setRemoveMode(false);
             }
          });
       removeButton.addActionListener(new ActionListener()
          {
             public void actionPerformed(ActionEvent event)
             {
                report.setRemoveMode(true);
             }
          });
       // layout components
       JPanel gridPanel = new JPanel();
       gridPanel.setLayout(new GridLayout(0, 2));
       gridPanel.add(new JLabel("Rental Record"));
       gridPanel.add(rentalRecord);
       gridPanel.add(new JLabel("Type of Car"));
       gridPanel.add(carType);
       gridPanel.add(new JLabel("Operation"));
       gridPanel.add(buttonPanel);
       add(gridPanel, BorderLayout.NORTH);
```

```
        add(carComponent, BorderLayout.CENTER);
    }
    private JTextField rentalRecord;
    private JComboBox carType;
    private JComponent carComponent;
    private JRadioButton addButton;
    private JRadioButton removeButton;
    private DamageReport report;
    private JFileChooser chooser;
    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 400;
    private static Map<DamageReport.CarType, Shape> shapes =
     new EnumMap<DamageReport.CarType,
Shape>(DamageReport.CarType.class);
    static
   {
      int width = 200;
      int x = 50;
      int y = 50;
      Rectangle2D.Double body = new Rectangle2D.Double(x, y + width /
6, width - 1, width / 6);
     Ellipse2D.Double frontTire = new Ellipse2D.Double(x + width / 6,
y + width / 3,
          width / 6, width / 6);
      Ellipse2D.Double rearTire = new Ellipse2D.Double(x + width * 2
/ 3, y + width / 3,
           width / 6, width / 6);
      Point2D.Double p1 = new Point2D.Double(x + width / 6, y + width
/ 6);
      Point2D.Double p2 = new Point2D.Double(x + width / 3, y);
      Point2D.Double p3 = new Point2D.Double(x + width * 2 / 3, y);
      Point2D.Double p4 = new Point2D.Double(x + width * 5 / 6, y +
width / 6);
      Line2D.Double frontWindshield = new Line2D.Double(p1, p2);
      Line2D.Double roofTop = new Line2D.Double(p2, p3);
      Line2D.Double rearWindshield = new Line2D.Double(p3, p4);
      GeneralPath sedanPath = new GeneralPath();
      sedanPath.append(frontTire, false);
      sedanPath.append(rearTire, false);
      sedanPath.append(body, false);
      sedanPath.append(frontWindshield, false);
      sedanPath.append(roofTop, false);
      sedanPath.append(rearWindshield, false);
      shapes.put(DamageReport.CarType.SEDAN, sedanPath);
      Point2D.Double p5 = new Point2D.Double(x + width * 11 / 12, y);
```

```
      Point2D.Double p6 = new Point2D.Double(x + width, y + width /
6);
      roofTop = new Line2D.Double(p2, p5);
      rearWindshield = new Line2D.Double(p5, p6);
      GeneralPath wagonPath = new GeneralPath();
      wagonPath.append(frontTire, false);
      wagonPath.append(rearTire, false);
      wagonPath.append(body, false);
      wagonPath.append(frontWindshield, false);
      wagonPath.append(roofTop, false);
      wagonPath.append(rearWindshield, false);
      shapes.put(DamageReport.CarType.WAGON, wagonPath);
      Point2D.Double p7 = new Point2D.Double(x + width / 3, y - width
/ 6);
      Point2D.Double p8 = new Point2D.Double(x + width * 11 / 12, y -
width / 6);
      frontWindshield = new Line2D.Double(p1, p7);
      roofTop = new Line2D.Double(p7, p8);
      rearWindshield = new Line2D.Double(p8, p6);
      GeneralPath suvPath = new GeneralPath();
      suvPath.append(frontTire, false);
      suvPath.append(rearTire, false);
      suvPath.append(body, false);
      suvPath.append(frontWindshield, false);
      suvPath.append(roofTop, false);
      suvPath.append(rearWindshield, false);
       shapes.put(DamageReport.CarType.SUV, suvPath);
   }
}
```

DamageReport.java

```
import java.awt.*;
import java.awt.geom.*;
import java.beans.*;
import java.util.*;
 /**
* This class describes a vehicle damage report that will be saved and
loaded with the
* long-term persistence mechanism.
* @version 1.21 2004-08-30
* @author Cay Horstmann
*/
public class DamageReport
```

```
{
public enum CarType
    {
      SEDAN, WAGON, SUV
}
// this property is saved automatically
public void setRentalRecord(String newValue)
{
      rentalRecord = newValue;
}
public String getRentalRecord()
     {
        return rentalRecord;
}
// this property is saved automatically
public void setCarType(CarType newValue)
    {
       carType = newValue;
    }
public CarType getCarType()
 {
    return carType;
   }
// this property is set to be transient
public void setRemoveMode(boolean newValue)
{
       removeMode = newValue;
}
public boolean getRemoveMode()
   {
       return removeMode;
}
public void click(Point2D p)
  {
      if (removeMode)
      {
         for (Point2D center : points)
         {
             Ellipse2D circle = new Ellipse2D.Double(center.getX() -
MARK_SIZE, center.getY()
                 - MARK_SIZE, 2 * MARK_SIZE, 2 * MARK_SIZE);
          if (circle.contains(p))
           {
              points.remove(center);
       return;
```

```
            }
          }
    }
        else points.add(p);
}
 public void drawDamage(Graphics2D g2)
    {
      g2.setPaint(Color.RED);
      for (Point2D center : points)
{
    Ellipse2D circle = new Ellipse2D.Double(center.getX() -
MARK_SIZE, center.getY()
               - MARK_SIZE, 2 * MARK_SIZE, 2 * MARK_SIZE);
        g2.draw(circle);
      }
}
public void configureEncoder(XMLEncoder encoder)
{
// this step is necessary to save Point2D.Double objects
    encoder.setPersistenceDelegate(Point2D.Double.class, new
DefaultPersistenceDelegate(
         new String[] { "x", "y" }));
// this step is necessary because the array list of points is not
 // (and should not be) exposed as a property
       encoder.setPersistenceDelegate(DamageReport.class, new
DefaultPersistenceDelegate()
          {
                protected void initialize(Class<?> type, Object
oldInstance, Object newInstance,
                Encoder out)
          {
               super.initialize(type, oldInstance, newInstance, out);
               DamageReport r = (DamageReport) oldInstance;
               for (Point2D p : r.points)
                out.writeStatement(new Statement(oldInstance,
"click", new Object[] { p }));
             }
          });
    }
 // this step is necessary to make the removeMode property transient
    static
    {
       try
      {
          BeanInfo info =
```

```
Introspector.getBeanInfo(DamageReport.class);
          for (PropertyDescriptor desc :
info.getPropertyDescriptors())
            if (desc.getName().equals("removeMode"))
desc.setValue("transient", Boolean.TRUE);
       }
       catch (IntrospectionException e)
       {
          e.printStackTrace();
       }
   }
    private String rentalRecord;
    private CarType carType;
    private boolean removeMode;
    private ArrayList<Point2D> points = new ArrayList<Point2D>();
    private static final int MARK_SIZE = 5;
}
```

java.beans.XMLEncoder 1.4

- XMLEncoder(OutputStream out)

constructs an XMLEncoder that sends its output to the given stream.

- void writeObject(Object obj)

archives the given object.

- void writeStatement(Statement stat)

writes the given statement to the archive. This method should only be called from a persistence delegate.

java.beans.Encoder 1.4

- void setPersistenceDelegate(Class<?> type, PersistenceDelegate delegate)
- PersistenceDelegate getPersistenceDelegate(Class<?> type)

sets or gets the delegate for archiving objects of the given type.

- void setExceptionListener(ExceptionListener listener)
- ExceptionListener getExceptionListener()

sets or gets the exception listener that is notified if an exception occurs during the encoding process.

java.beans.ExceptionListener 1.4

- void exceptionThrown(Exception e)

is called when an exception was thrown during the encoding or decoding process.

java.beans.XMLDecoder 1.4

- XMLDecoder(InputStream in)

constructs an XMLDecoder that reads an archive from the given input stream.

- Object readObject()

reads the next object from the archive.

- void setExceptionListener(ExceptionListener listener)
- ExceptionListener getExceptionListener()

sets or gets the exception listener that is notified if an exception occurs during the encoding process.

java.beans.PersistenceDelegate 1.4

- protected abstract Expression instantiate(Object oldInstance, Encoder out)

returns an expression for instantiating an object that is equivalent to oldInstance.

- protected void initialize(Class<?> type, Object oldInstance, Object newInstance, Encoder out)

writes statements to out that turn newInstance into an object that is equivalent to oldInstance.

java.beans.DefaultPersistenceDelegate 1.4

- DefaultPersistenceDelegate()

constructs a persistence delegate for a class with a zero-parameter constructor.

- DefaultPersistenceDelegate(String[] propertyNames)

constructs a persistence delegate for a class whose construction parameters are the values of the given properties.

- protected Expression instantiate(Object oldInstance, Encoder out)

returns an expression for invoking the constructor with either no parameters or the values of the properties specified in the constructor.

- protected void initialize(Class<?> type, Object oldInstance, Object newInstance, Encoder out)

writes statements to out that apply property setters to newInstance, attempting to turn it into an object that is equivalent to oldInstance.

java.beans.Expression 1.4

- Expression(Object value, Object target, String methodName, Object[] parameters)

constructs an expression that calls the given method on target, with the given parameters. The result of the expression is assumed to be value. To call a constructor, target should be a Class

object and methodName should be "new".

java.beans.Statement 1.4

- Statement(Object target, String methodName, Object[] parameters)

constructs a statement that calls the given method on target, with the given parameters.

9 Distributed Objects

This section discusses an overview of Distributed Objects and the code demonstration.

9.1 Overview of Distributed Objects

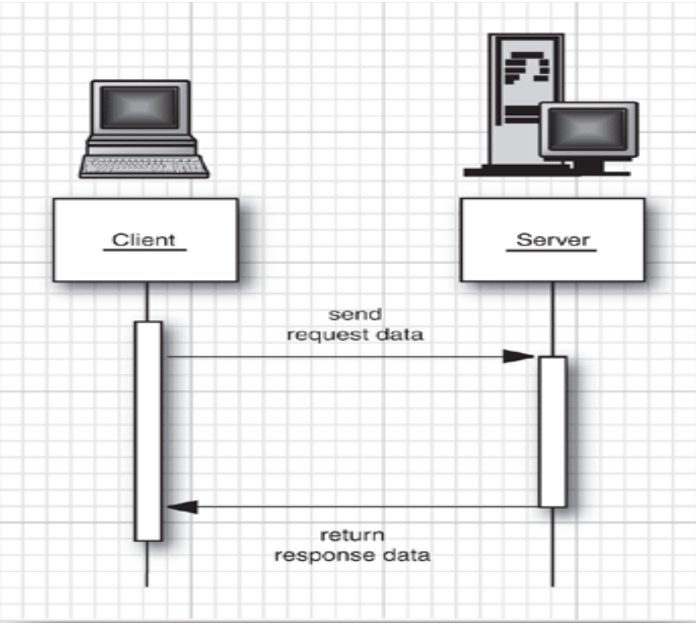
Periodically, the programming community starts thinking of "objects everywhere" as the solution to all its problems. The idea is to have a happy family of collaborating objects that can be located anywhere. When an object on one computer needs to invoke a method on an object on another computer, it sends a network message that contains the details of the request. The remote object computes a response, perhaps by accessing a database or by communicating with additional objects. Once the remote object has the answer to the client request, it sends the answer back over the network. Conceptuatlly, this process sounds quite simple, but you need to understand what goes on under the hood to use distributed objects effectively.

In this chapter, we focus on Java technologies for distributed programming, in particular the Remote Method Invocation (RMI) protocol for communicating between two Java virtual machines (which might run on different computers). We then briefly visit the JAX-WS technology for making remote calls to web services.

The Roles of Client and Server

The basic idea behind all distributed programming is simple. A client computer makes a request and sends the request data across a network to a server. The server processes the request and sends back a response for the client to analyze. Figure 36 shows the process.

Figure 36. Transmitting objects between client and server

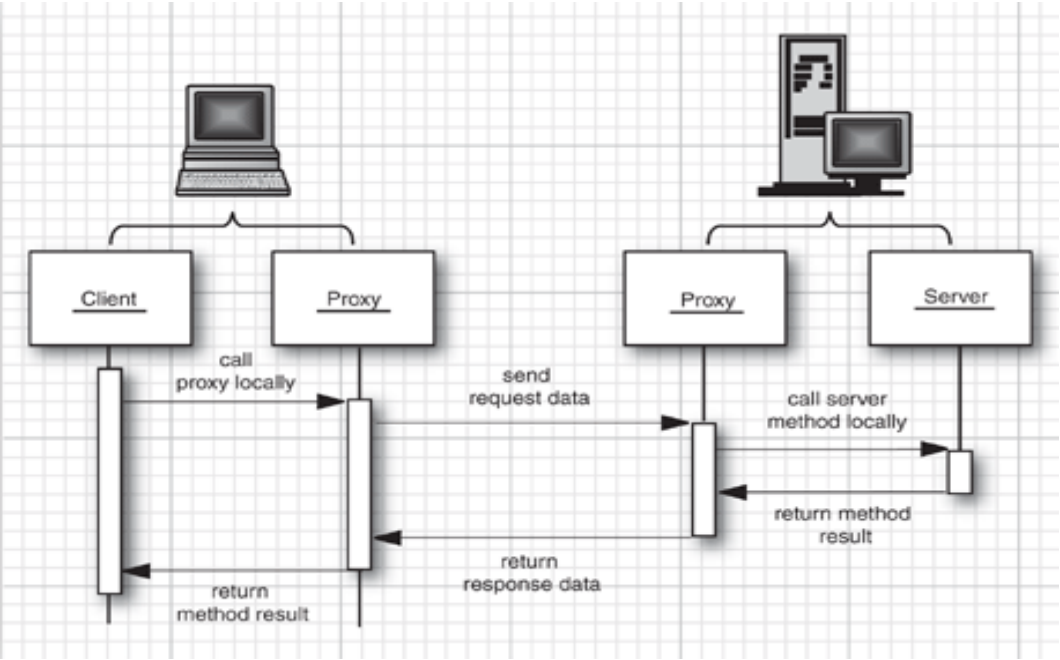


We would like to say at the outset that these requests and responses are not what you would see in a web application. The client is not a web browser. It can be any application that executes business rules of any complexity. The client application might or might not interact with a human user, and if it does, it can have a command-line or Swing user interface. The protocol for the request and response data allows the transfer of arbitrary objects, whereas traditional web applications are limited by using HTTP for the request and HTML for the response.

What we want is a mechanism by which the client programmer makes a regular method call, without worrying about sending data across the network or parsing the response. The solution is to install a proxy object on the client. The proxy is an object located in the client virtual machine that appears to the client program as if it was the remote object. The client calls the proxy, making a regular method call. The client proxy contacts the server, using a network protocol.

Similarly, the programmer who implements the service doesn't want to fuss with client communication. The solution is to install a second proxy object on the server. The server proxy communicates with the client proxy, and it makes regular method calls to the object implemeting the service (see Figure 37).

Figure 37. Remote method call with proxies



How do the proxies communicate with each other? That depends on the implementation technology. There are three common choices:

- The Java RMI technology supports method calls between distributed Java objects.
- The Common Object Request Broker Architecture (CORBA) supports method calls between objects of any programming language. CORBA uses the binary Internet Inter-ORB Protocol, or IIOP, to communicate between objects.
- The web services architecture is a collection of protocols, sometimes collectively described as WS-*. It is also programming-language neutral. However, it uses XML-based communication formats. The format for transmitting objects is the Simple Object Access Protocol (SOAP).

If the communicating programs are implemented in Java code, then the full generality and complexity of CORBA or WS-* is not required. Sun developed a simple mechanism, called RMI, specifically for communication between Java applications.

It is well worth learning about RMI, even if you are not going to use it in your own programs. You will learn the mechanisms that are essential for programming distributed applications, using

a straightforward architecture. Moreover, if you use enterprise Java technologies, it is very useful to have a basic understanding of RMI because that is the protocol used to communicate between enterprise Java beans (EJBs). EJBs are server-side components that are composed to make up complex applications that run on multiple servers. To make effective use of EJBs, you will want to have a good idea of the costs associated with remote calls.

Unlike RMI, CORBA and SOAP are completely language neutral. Client and server programs can be written in C, C++, C#, Java, or any other language. You supply an interface description to specify the signatures of the methods and the types of the data your objects can handle. These descriptions are formatted in a special language, called Interface Definition Language (IDL) for CORBA and Web Services Description Language (WSDL) for web services.

For many years, quite a few people believed that CORBA was the object model of the future. Frankly, though, CORBA has a reputation—sometimes deserved—for complex implementations and interoperability problems, and it has only reached modest success. We covered interoperability between Java and CORBA for five editions of this book, but dropped it for lack of interest. Our sentiments about CORBA are similar to those expressed by French president Charles De Gaulle about Brazil: It has a great future . . . and always will.

Web services had a similar amount of buzz when they first appeared, with the promise that they are simpler and, of course, founded in the goodness of the World Wide Web and XML. However, with the passing of time and the work of many committees, the protocol stack has become less simple, as it acquired more of the features that CORBA had all along. The XML protocol has the advantage of being (barely) human-readable, which helps with debugging. On the other hand, XML processing is a significant performance bottleneck. Recently, the WS-* stack has lost quite a bit of its luster and it too is gaining a reputation—sometimes deserved—for complex implementations and interoperability problems.

We close this chapter with an example of an application that consumes a web service. We have a look at the underlying protocol so that you can see how communication between different programming languages is implemented.

Remote Method Calls

The key to distributed computing is the remote method call. Some code on one machine (called the client) wants to invoke a method on an object on another machine (the remote object). To make this possible, the method parameters must somehow be shipped to the other machine, the server must be informed to locate the remote object and execute the method, and the return value must be shipped back.

Before looking at this process in detail, we want to point out that the client/server terminology applies only to a single method call. The computer that calls the remote method is the client for that call, and the computer hosting the object that processes the call is the server for that call. It is entirely possible that the roles are reversed somewhere down the road. The server of a previous call can itself become the client when it invokes a remote method on an object residing on another computer.

Stubs and Parameter Marshalling

When client code wants to invoke a method on a remote object, it actually calls an ordinary method on a proxy object called a stub. For example,

Warehouse centralWarehouse = get stub object;

double price = centralWarehouse.getPrice("Blackwell Toaster");

The stub resides on the client machine, not on the server. It knows how to contact the server over the network. The stub packages the parameters used in the remote method into a block of bytes. The process of encoding the parameters is called parameter marshalling. The purpose of parameter marshalling is to convert the parameters into a format suitable for transport from one virtual machine to another. In the RMI protocol, objects are encoded with the serialization mechanism that is described in Chapter 1. In the SOAP protocol, objects are encoded as XML.

To sum up, the stub method on the client builds an information block that consists of

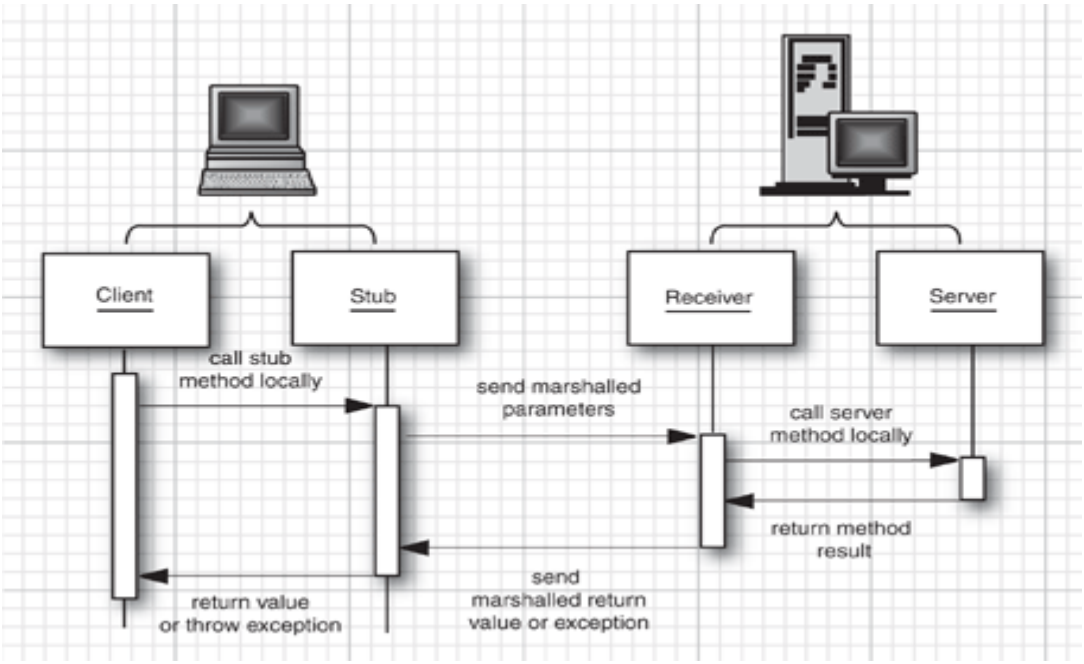
- An identifier of the remote object to be used.
- A description of the method to be called.
- The parameters.

The stub then sends this information to the server. On the server side, a receiver object performs the following actions:

1. It locates the remote object to be called.
2. It calls the desired method, passing the supplied parameters.
3. It captures the return value or exception of the call.
4. It sends a package consisting of the marshalled return data back to the stub on the client.

The client stub unmarshals the return value or exception from the server. This value becomes the return value of the stub call. Or, if the remote method threw an exception, the stub rethrows it in the virtual machine of the caller. Figure 38 shows the information flow of a remote method invocation.

Figure 38. Parameter marshalling



This process is obviously complex, but the good news is that it is completely automatic and, to a large extent, transparent for the programmer. The details for implementing remote objects and

for getting client stubs depend on the technology for distributed objects. In the following sections, we have a close look at RMI.

The RMI Programming Model

To introduce the RMI programming model, we start with a simple example. A remote object represents a warehouse. The client program asks the warehouse about the price of a product. In the following sections, you will see how to implement and launch the server and client programs.

Interfaces and Implementations

The capabilities of remote objects are expressed in interfaces that are shared between the client and server. For example, the interface in *Warehouse.java* describes the service provided by a remote warehouse object:

Interfaces for remote objects must always extend the Remote interface defined in the java.rmi package. All the methods in those interfaces must also declare that they will throw a *RemoteException*. Remote method calls are inherently less reliable than local calls—it is always possible that a remote call will fail. For example, the server might be temporarily unavailable, or there might be a network problem. Your client code must be prepared to deal with these possibilities. For these reasons, you must handle the *RemoteException* with every remote method call and specify the appropriate action to take when the call does not succeed.

Next, on the server side, you must provide the class that actually carries out the work advertised in the remote interface—see *Warehouseimpl.java*.

The *WarehouseImpl* constructor is declared to throw a *RemoteException* because the superclass constructor can throw that exception. This happens when there is a problem connecting to the network service that tracks remote objects.

You can tell that the class is the target of remote method calls because it extends *UnicastRemoteObject*. The constructor of that class makes objects remotely accessible. The "path of least resistance" is to derive from *UnicastRemoteObject*, and all service implementation classes in this chapter do so.

Occasionally, you might not want to extend the *UnicastRemoteObject* class, perhaps because your implementation class already extends another class. In that situation, you need to manually instantiate the remote objects and pass them to the static exportObject method. Instead of extending *UnicastRemoteObject*, call

UnicastRemoteObject.exportObject(this, 0);

in the constructor of the remote object. The second parameter is 0 to indicate that any suitable port can be used to listen to client connections.

The RMI Registry

To access a remote object that exists on the server, the client needs a local stub object. How can the client request such a stub? The most common method is to call a remote method of another remote object and get a stub object as a return value. There is, however, a chicken-and-egg problem here: The first remote object has to be located some other way. For that purpose, the JDK provides a bootstrap registry service.

A server program registers at least one remote object with a bootstrap registry. To register a remote object, you need a RMI URL and a reference to the implementation object.

RMI URLs start with rmi: and contain an optional host name, an optional port number, and the name of the remote object that is (hopefully) unique. An example is:

```
rmi://regserver.mycompany.com:99/central_warehouse
```

By default, the host name is localhost and the port number is 1099. The server tells the registry at the given location to associate or "bind" the name with the object.

Here is the code for registering a *WarehouseImpl* object with the RMI registry on the same server:

```
WarehouseImpl centralWarehouse = new WarehouseImpl();

Context namingContext = new InitialContext();
```

```
namingContext.bind("rmi:central_warehouse", centralWarehouse);
```

The program in *WarehouseServer.java* simply constructs and registers a *WarehouseImpl* object.

A client can enumerate all registered RMI objects by calling:

Enumeration<NameClassPair> e = namingContext.list("rmi://regserver.mycompany.com");

NameClassPair is a helper class that contains both the name of the bound object and the name of its class. For example, the following code displays the names of all registered objects:

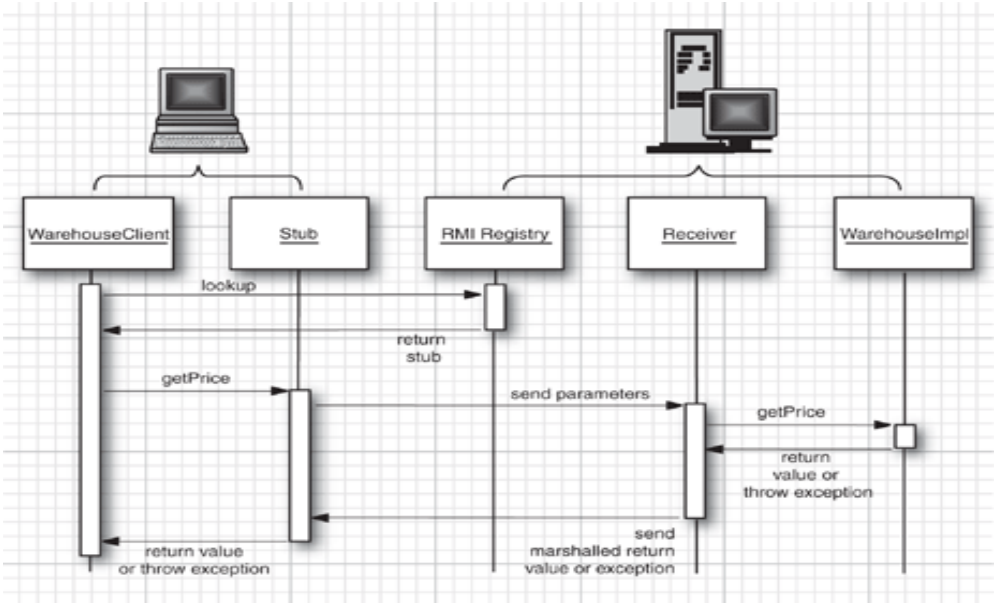
```
while (e.hasMoreElements())
   System.out.println(e.nextElement().getName());
```

A client gets a stub to access a remote object by specifying the server and the remote object name in the following way:

```
String url = "rmi://regserver.mycompany.com/central_warehouse";
Warehouse centralWarehouse = (Warehouse) namingContext.lookup(url);
```

The code in *WarehouseClient.java* shows the client that obtains a stub to the remote warehouse object and invokes the remote getPrice method. Figure 39 shows the flow of control. The client obtains a Warehouse stub and invokes the getPrice method on it. Behind the scenes, the stub contacts the server and causes the getPrice method to be invoked on the WarehouseImpl object.

Figure 39. Calling the remote getDescription method



Deploying the Program

Deploying an application that uses RMI can be tricky because so many things can go wrong and the error messages that you get when something does go wrong are so poor. We have found that it really pays off to test the deployment under realistic conditions, separating the classes for client and server.

Make two separate directories to hold the classes for starting the server and client.

```
server/
   WarehouseServer.class
   Warehouse.class
   WarehouseImpl.class
client/
   WarehouseClient.class
   Warehouse.class
```

When deploying RMI applications, one commonly needs to dynamically deliver classes to running programs. One example is the RMI registry. Keep in mind that one instance of the registry will serve many different RMI applications. The RMI registry needs to have access to the class files of the service interfaces that are being registered. When the registry starts, however, one cannot predict all future registration requests. Therefore, the RMI registry dynamically loads the class files of any remote interfaces that it has not previously encountered.

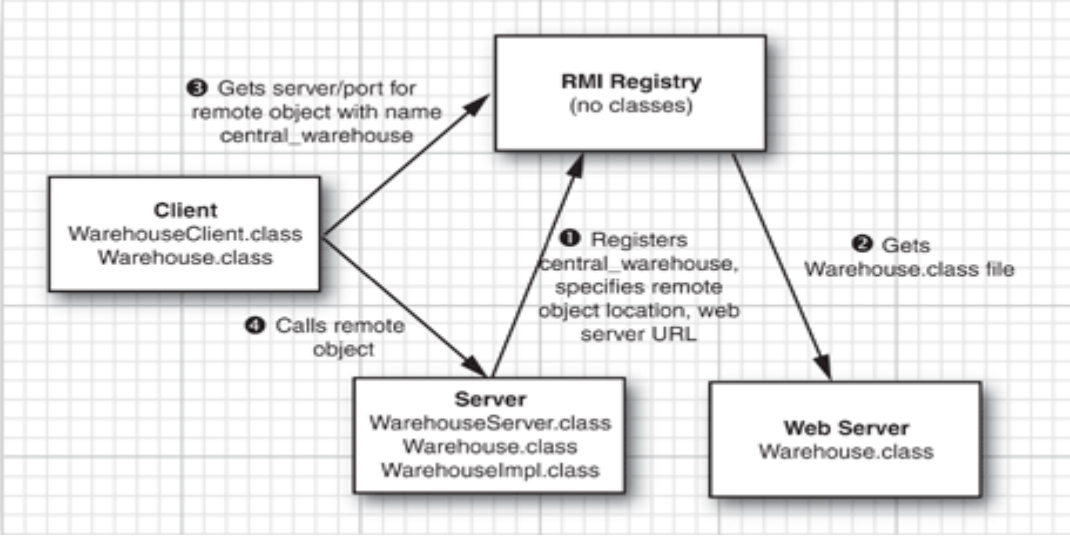
Dynamically delivered class files are distributed through standard web servers. In our case, the server program needs to make the Warehouse.class file available to the RMI registry, so we put that file into a third directory that we call download.

```
download/

   Warehouse.class
```

We use a web server to serve the contents of that directory.

When the application is deployed, the server, RMI registry, web server, and client can be located on four different computers—see Figure 40. However, for testing purposes, we will use a single computer.

Figure 40. Server calls in the Warehouse application



To test the sample application, use the *NanoHTTPD* web server that is available from http://elonen.iki.fi/code/nanohttpd. This tiny web server is implemented in a single Java source

file. Open a new console window, change to the download directory, and copy *NanoHTTPD.java* to that directory. Compile the source file and start the web server, using the command

```
java NanoHTTPD 8080
```

The command-line argument is the port number. Use any other available port if port 8080 is already used on your machine.

Next, open another console window, change to a directory that contains no class files, and start the RMI registry:

```
Rmiregistry
```

Now you are ready to start the server. Open a third console window, change to the server directory, and issue the command

```
java -Djava.rmi.server.codebase=http://localhost:8080/ WarehouseServer
```

The *java.rmi.server.codebase* property points to the URL for serving class files. The server program communicates this URL to the RMI registry.

Have a peek at the console window running *NanoHTTPD*. You will see a message that demonstrates that the *Warehouse.class* file has been served to the RMI registry.

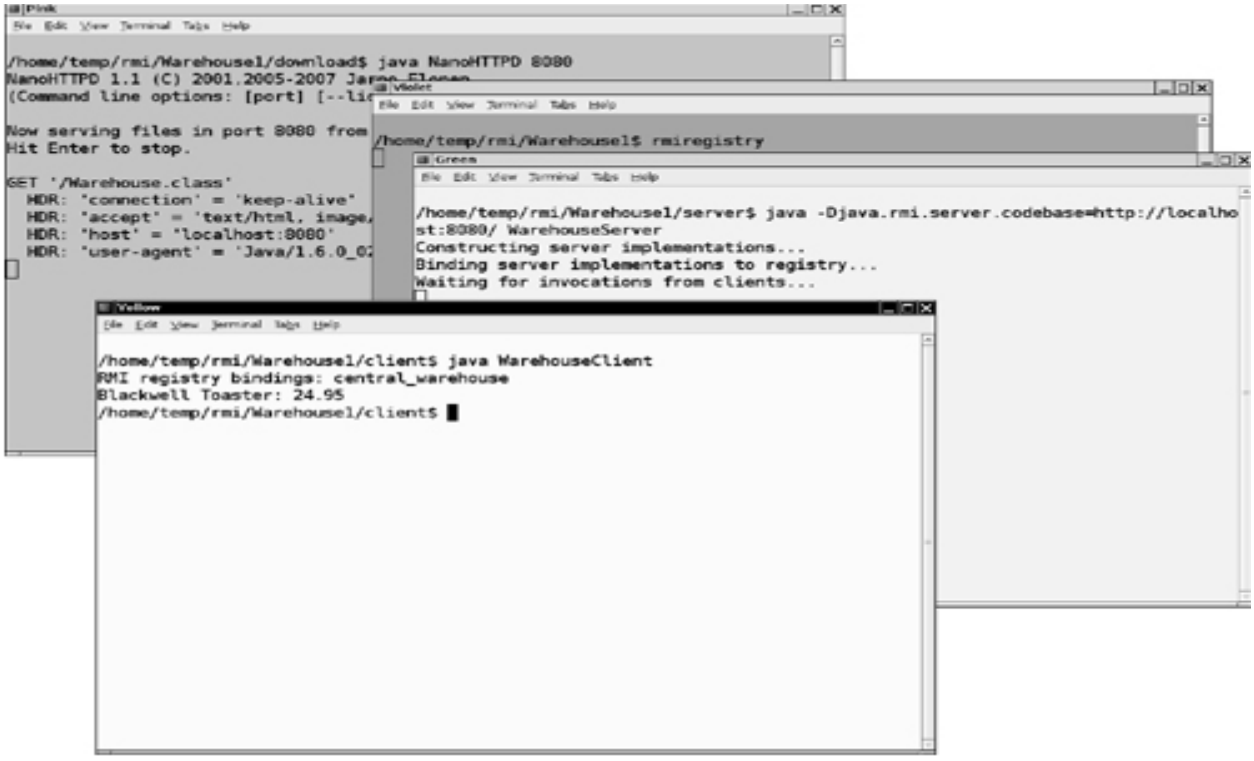
Note that the server program does not exit. This seems strange—after all, the program just creates a *WarehouseImpl* object and registers it. Actually, the main method does exit immediately after registration, as you would expect. However, when you create an object of a class that extends *UnicastRemoteObject*, a separate thread that keeps the program alive indefinitely is started. Thus, the program stays around to allow clients to connect to it.

Finally, open a fourth console window, change to the client directory, and run

```
java WarehouseClient
```

You will see a short message, indicating that the remote method was successfully invoked (see Figure 41).

Figure 41. Testing an RMI application



Logging RMI Activity

If you start the server with the option

```
Djava.rmi.server.logCalls=true WarehouseServer &
```

then the server logs all remote method calls on its console. Try it—you'll get a good impression of the RMI traffic.

If you want to see additional logging messages, you have to configure RMI loggers, using the standard Java logging API. (See Volume I, Chapter 11 for more information on logging.)

Make a file logging.properties with the following content:

```
handlers=java.util.logging.ConsoleHandler

.level=FINE

java.util.logging.ConsoleHandler.level=FINE
```

```
java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFor
matter
```

You can fine-tune the settings by setting individual levels for each logger rather than setting the global level. Figure 42 lists the RMI loggers. For example, to track the class loading activity, you can set

sun.rmi.loader.level=FINE

Figure 42. RMI Loggers

Logger Name	Logged Activity
sun.rmi.server.call	Server-side remote calls
sun.rmi.server.ref	Server-side remote references
sun.rmi.client.call	Client-side remote calls
sun.rmi.client.ref	Client-side remote references
sun.rmi.dgc	Distributed garbage collection
sun.rmi.loader	RMIClassLoader
sun.rmi.transport.misc	Transport layer
sun.rmi.transport.tcp	TCP binding and connection
sun.rmi.transport.proxy	HTTP tunneling

Start the RMI registry with the option

```
J-Djava.util.logging.config.file=directory/logging.properties
```

Start the client and server with

```
Djava.util.logging.config.file=directory/logging.properties
```

Here is an example of a logging message that shows a class loading problem: The RMI registry cannot find the *Warehouse* class because the web server has been shut down.

```
FINE: RMI TCP Connection(1)-127.0.1.1: (port 1099) op = 80

Oct 13, 2007 4:43:30 PM sun.rmi.server.LoaderHandler loadProxyClass
```

```
FINE: RMI TCP Connection(1)-127.0.1.1: interfaces = [java.rmi.Remote,
Warehouse], codebase = "http://localhost:8080/"

Oct 13, 2007 4:43:30 PM sun.rmi.server.LoaderHandler loadProxyClass

FINE: RMI TCP Connection(1)-127.0.1.1: proxy class resolution failed

java.lang.ClassNotFoundException: Warehouse
```

Parameters and Return Values in Remote Methods

At the start of a remote method invocation, the parameters need to be moved from the virtual machine of the client to the virtual machine of the server. After the invocation has completed, the return value needs to be transferred in the other direction. When a value is passed from one virtual machine to another other, we distinguish two cases: passing remote objects and passing nonremote objects. For example, suppose that a client of the WarehouseServer passes a Warehouse reference (that is, a stub through which the remote warehouse object can be called) to another remote method. That is an example of passing a remote object. However, most method parameters will be ordinary Java objects, not stubs to remote objects. An example is the String parameter of the getPrice method in our first sample application.

Transferring Remote Objects

When a reference to a remote object is passed from one virtual machine to the other, the sender and recipient of the remote object both hold a reference to the same entity. That reference is not a memory location (which is only meaningful in a single virtual machine), but it consists of a network address and a unique identifier for the remote object. This information is encapsulated in a stub object.

Conceptually, passing a remote reference is quite similar to passing local object references within a virtual machine. However, always keep in mind that a method call on a remote reference is significantly slower and potentially less reliable than a method call on a local reference.

Transferring Nonremote Objects

Consider the String parameter of the getPrice method. The string value needs to be copied from the client to the server. It is not difficult to imagine how a copy of a string can be transported

across a network. The RMI mechanism can also make copies of more complex objects, provided they are serializable. RMI uses the serialization mechanism described in Chapter 1 to send objects across a network connection. This means that any classes that implement the Serializable interface can be used as parameter or return types.

Passing parameters by serializing them has a subtle effect on the semantics of remote methods. When you pass objects into a local method, object references are transferred. When the method applies a mutator method to a parameter object, the caller will observe that change. But if a remote method mutates a serialized parameter, it changes the copy, and the caller will never notice.

To summarize, there are two mechanisms for transferring values between virtual machines.

- Objects of classes that implement the Remote interface are transferred as remote references.
- Objects of classes that implement the Serializable interface but not the Remote interface are copied using serialization.

All of this is automatic and requires no programmer intervention. Keep in mind that serialization can be slow for large objects, and that the remote method cannot mutate serialized parameters. You can, of course, avoid these issues by passing around remote references. That too comes at a cost: Invoking methods on remote references is far more expensive than calling local methods. Being aware of these costs allows you to make informed choices when designing remote services.

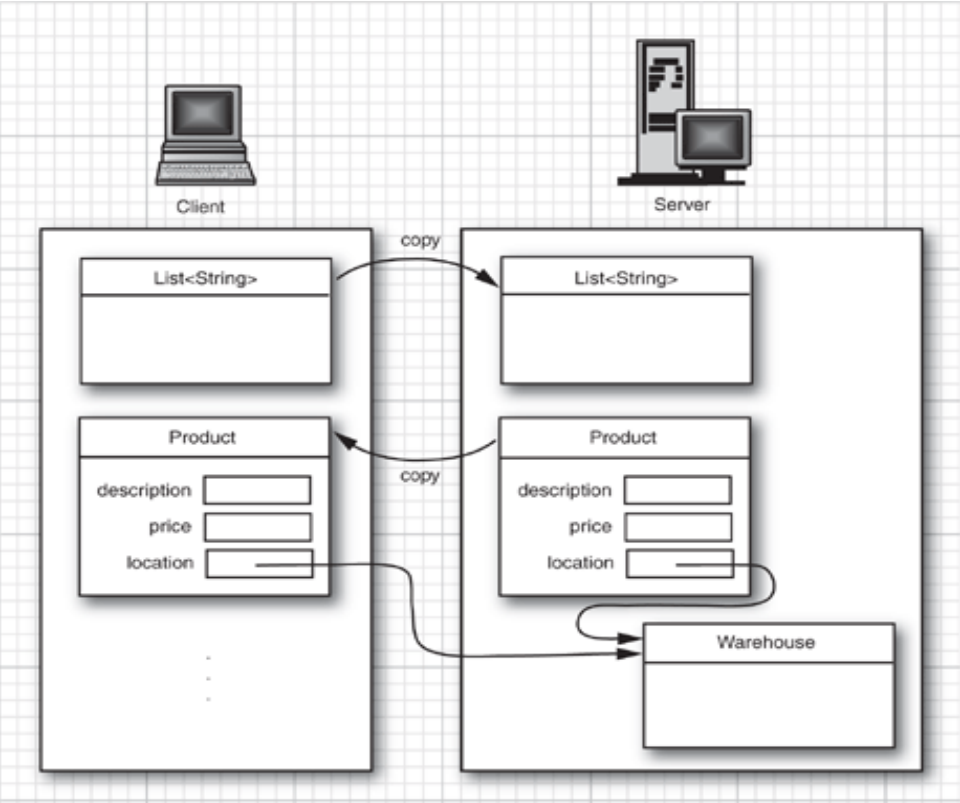
Our next example program will illustrate the transfer of remote and serializable objects. We change the Warehouse interface as shown in *Warehouse.java*. Given a list of keywords, the warehouse returns the Product that is the best match.

The parameter of the getProduct method has type List<String>. A parameter value must belong to a serializable class that implements the List<String> interface, such as ArrayList<String>. (Our sample client passes a value that is obtained by a call to Arrays.asList. Fortunately, that method is guaranteed to return a serializable list as well.)

The return type Product encapsulates the description, price, and location of the product—see *Product.java*.

Note that the Product class is serializable. The server constructs a Product object, and the client gets a copy (see Figure 43).

Figure 43. Copying local parameter and result objects



However, there is a subtlety. The Product class has an instance field of type Warehouse, a remote interface. The warehouse object is not serialized, which is just as well as it might have a huge amount of state. Instead, the client receives a stub to a remote Warehouse object. That stub might be different from the centralWarehouse stub on which the getProduct method was called. In our implementation, we will have two kinds of products, toasters and books, that are located in different warehouses.

Dynamic Class Loading

There is another subtlety to our next sample program. A list of keyword strings is sent to the server, and the warehouse returns an instance of a class Product. Of course, the client program

will need the class file Product.class to compile. However, whenever our server program cannot find a match for the keywords, it returns the one product that is sure to delight everyone: the Core Java book. That object is an instance of the Book class, a subclass of Product.

When the client was compiled, it might have never seen the Book class. Yet when it runs, it needs to be able to execute Book methods that override Product methods. This demonstrates that the client needs to have the capability of loading additional classes at runtime. The client uses the same mechanism as the RMI registry. Classes are served by a web server, the RMI server class communicates the URL to the client, and the client makes an HTTP request to download the class files.

Whenever a program loads new code from another network location, there is a security issue. For that reason, you need to use a security manager in RMI applications that dynamically load classes. (See Chapter 9 for more information on class loaders and security managers.)

Programs that use RMI should install a security manager to control the activities of the dynamically loaded classes. You install it with the instruction

```
System.setSecurityManager(new SecurityManager());
```

By default, the *SecurityManager* restricts all code in the program from establishing network connections. However, the program needs to make network connections to three remote locations:

- The web server that loads remote classes.
- The RMI registry.
- Remote objects.

To allow these operations, you supply a policy file. (We discussed policy files in greater detail in Chapter 9.) Here is a policy file that allows an application to make any network connection to a port with port number of at least 1024. (The RMI port is 1099 by default, and the remote objects also use ports 1024. We use port 8080 for dowloading classes.)

grant

```
{
   permission java.net.SocketPermission
      "*:1024-65535", "connect";
};
```

You need to instruct the security manager to read the policy file by setting the java.security.policy property to the file name. You can use a call such as

```
System.setProperty("java.security.policy", "rmi.policy");
```

Alternatively, you can specify the system property setting on the command line:

```
Djava.security.policy=rmi.policy
```

To run the sample application, be sure that you have killed the RMI registry, web server, and the server program from the preceding sample. Open four console windows and follow these steps.

1. Compile the source files for the interface, implementation, client, and server classes.

   ```
   javac *.java
   ```

2. Make three directories, client, server, and download, and populate them as follows:

   ```
   client/
      WarehouseClient.class
      Warehouse.class
      Product.class
      client.policy
   server/
      Warehouse.class
      Product.class
      Book.class
   ```

```
      WarehouseImpl.class
      WarehouseServer.class
      server.policy
   download
      Warehouse.class
      Product.class
      Book.class
```

3. In the first console window, change to a directory that has no class files. Start the RMI registry.

4. In the second console window, change to the download directory and start *NanoHTTPD*.

5. In the third console window, change to the server directory and start the server.

```
java -Djava.rmi.server.codebase=http://localhost:8080/
WarehouseServer
```

6. In the fourth console window, change to the client directory and run the client.

```
java WarehouseClient
```

Book.java shows the code of the Book class. Note that the *getDescription* method is overridden to show the ISBN. When the client program runs, it shows the ISBN for the Core Java book, which proves that the Book class was loaded dynamically. *WarehouseImpl.java (parameter and return* shows the warehouse implementation. A warehouse has a reference to a backup warehouse. If an item cannot be found in the warehouse, the backup warehouse is searched. *WarehouseServer.java (parameter and return section)* shows the server program. Only the central warehouse is entered into the RMI registry. Note that a remote reference to the backup warehouse can be passed to the client even though it is not included in the RMI registry. This happens whenever no keyword matches and a Core Java book (whose location field references the backup warehouse) is sent to the client.

Remote References with Multiple Interfaces

A remote class can implement multiple interfaces. Consider a remote interface *ServiceCenter*.

```
public interface ServiceCenter extends Remote
{
   int getReturnAuthorization(Product prod) throws RemoteException;
}
```

Now suppose a *WarehouseImpl* class implements this interface as well as the Warehouse interface. When a remote reference to such a service center is transferred to another virtual machine, the recipient obtains a stub that has access to the remote methods in both the ServiceCenter and the Warehouse interface. You can use the instanceof operator to find out whether a particular remote object implements an interface. Suppose you receive a remote object through a variable of type Warehouse.

```
Warehouse location = product.getLocation();
```

The remote object might or might not be a service center. To find out, use the test

```
if (location instanceof ServiceCenter)
```

If the test passes, you can cast location to the *ServiceCenter* type and invoke the *getReturnAuthorization* method.

Remote Objects and the equals, hashCode, and clone Methods

Objects inserted in sets must override the equals method. In the case of a hash set or hash map, the hashCode method must be defined as well. However, there is a problem when trying to compare remote objects. To find out if two remote objects have the same contents, the call to equals would need to contact the servers containing the objects and compare their contents. Like any remote call, that call could fail. But the equals method in the class Object is not declared to throw a RemoteException, whereas all methods in a remote interface must throw that exception. Because a subclass method cannot throw more exceptions than the superclass method it replaces, you cannot define an equals method in a remote interface. The same holds for hashCode.

Instead, the equals and hashCode methods on stub objects simply look at the location of the remote objects. The equals method deems two stubs equal if they refer to the same remote object. Two stubs that refer to different remote objects are never equal, even if those objects have identical contents. Similarly, the hash code is computed only from the object identifier.

For the same technical reasons, remote references do not have a clone method. If clone were to make a remote call to tell the server to clone the implementation object, then the clone method would need to throw a *RemoteException*. However, the clone method in the Object superclass promised never to throw any exception other than *CloneNotSupportedException*.

To summarize, you can use remote references in sets and hash tables, but you must remember that equality testing and hashing do not take into account the contents of the remote objects. You simply cannot clone remote references.

Remote Object Activation

In the preceding sample programs, we used a server program to instantiate and register objects so that clients could make remote calls on them. However, in some cases, it might be wasteful to instantiate lots of remote objects and have them wait for connections, whether or not client objects use them. The activation mechanism lets you delay the object construction so that a remote object is only constructed when at least one client invokes a remote method on it.

To take advantage of activation, the client code is completely unchanged. The client simply requests a remote reference and makes calls through it.

However, the server program is replaced by an activation program that constructs activation descriptors of the objects that are to be constructed at a later time, and binds receivers for remote method calls with the naming service. When a call is made for the first time, the information in the activation descriptor is used to construct the object.

A remote object that is used in this way should extend the Activatable class instead of the *UnicastRemoteObject* class. Of course, it also implements one or more remote interfaces. For example,

```
class WarehouseImpl
```

```
   extends Activatable
   implements Warehouse
{
   . . .
}
```

Because the object construction is delayed until a later time, it must happen in a standardized form. Therefore, you must provide a constructor that takes two parameters:

- An activation ID (which you simply pass to the superclass constructor).
- A single object containing all construction information, wrapped in a *MarshalledObject*.

If you need multiple construction parameters, you must package them into a single object. You can always use an *Object[]* array or an ArrayList for this purpose.

When you build the activation descriptor, you will construct a *MarshalledObject* from the construction information like this:

```
MarshalledObject<T> param = new MarshalledObject<T>(constructionInfo);
```

In the constructor of the implementation object, use the get method of the *MarshalledObject* class to obtain the deserialized construction information.

```
T constructionInfo = param.get();
```

To demonstrate activation, we modify the *WarehouseImpl* class so that the construction information is a map of descriptions and prices. That information is wrapped into a *MarshalledObject* and unwrapped in the constructor:

Code View:

```
public WarehouseImpl(ActivationID id, MarshalledObject<Map<String,
Double>> param)
      throws RemoteException, ClassNotFoundException, IOException
{
```

```
   super(id, 0);
   prices = param.get();
   System.out.println("Warehouse implementation constructed.");
}
```

By passing 0 as the second parameter of the superclass constructor, we indicate that the RMI library should assign a suitable port number to the listener port.

This constructor prints a message so that you can see that the warehouse object is activated on demand.

Now let us turn to the activation program. First, you need to define an activation group. An activation group describes common parameters for launching the virtual machine that contains the remote objects. The most important parameter is the security policy.

Construct an activation group descriptor as follows:

```
Properties props = new Properties();
props.put("java.security.policy", "/path/to/server.policy");
ActivationGroupDesc group = new ActivationGroupDesc(props, null);
```

The second parameter describes special command options. We don't need any for this example, so we pass a null reference.

Next, create a group ID with the call

```
ActivationGroupID id =
ActivationGroup.getSystem().registerGroup(group);
```

Now you are ready to construct activation descriptors. For each object that should be constructed on demand, you need the following:

- The activation group ID for the virtual machine in which the object should be constructed.

- The name of the class (such as "*WarehouseImpl*" or "*com.mycompany.MyClassImpl*").
- The URL string from which to load the class files. This should be the base URL, not including package paths.
- The marshalled construction information.

For example,

```
MarshalledObject param = new MarshalledObject(constructionInfo);
ActivationDesc desc = new ActivationDesc(id, "WarehouseImpl",

"http://myserver.com/download/", param);
```

Pass the descriptor to the static Activatable.register method. It returns an object of some class that implements the remote interfaces of the implementation class. You can bind that object with the naming service:

```
Warehouse centralWarehouse = (Warehouse) Activatable.register(desc);
namingContext.bind("rmi:central_warehouse", centralWarehouse);
```

Unlike the server programs of the preceding examples, the activation program exits after registering and binding the activation receivers. The remote objects are constructed only when the first remote method call occurs.

WarehouseActivator.java and *WarehouseImpl.java (remote object activation)* show the code for the activation program and the activatable warehouse implementation. The warehouse interface and the client program are unchanged.

To launch this program, follow these steps:

1. Compile all source files.
2. Distribute class files as follows:

   ```
   client/
   ```

```
      WarehouseClient.class

      Warehouse.class

   server/

      WarehouseActivator.class

      Warehouse.class

      WarehouseImpl.class

      server.policy

   download/

      Warehouse.class

      WarehouseImpl.class

   rmi/

      rmid.policy
```

3. Start the RMI registry in the rmi directory (which contains no class files).

4. Start the RMI activation daemon in the rmi directory.

```
rmid -J-Djava.security.policy=rmid.policy
```

The rmid program listens to activation requests and activates objects in a separate virtual machine. To launch a virtual machine, the rmid program needs certain permissions. These are specified in a policy file (see *rmid.policy*). You use the -J option to pass an option to the virtual machine running the activation daemon.

5. Start the *NanoHTTPD* web server in the download directory.

6. Run the activation program from the server directory.

```
java -Djava.rmi.server.codebase=http://localhost:8080/
WarehouseActivator
```

The program exits after the activation receivers have been registered with the naming service. (You might wonder why you need to specify the codebase as it is also provided in the constructor of the activation descriptor. However, that information is only processed by the RMI activation daemon. The RMI registry still needs the codebase to load the remote interface classes.)

7. Run the client program from the client directory.

```
java WarehouseClient
```

The client will print the familiar product description. When you run the client for the first time, you will also see the constructor messages in the shell window of the activation daemon.

Web Services and JAX-WS

In recent years, web services have emerged as a popular technology for remote method calls. Technically, a web service has two components:

- A service that can be accessed with the SOAP transport protocol
- A description of the service in the WSDL format

SOAP is an XML protocol for invoking remote methods, similar to the protocol that RMI uses for the communication between clients and servers. Just as you can program RMI applications without knowing anything about the details of the RMI protocol, you don't really need to know any details about SOAP to call a web service.

WSDL is an interface description language. It too is based on XML. A WSDL document describes the interface of a web service: the methods that can be called, and their parameter and return types. In this section, we generate a WSDL document from a service implemented in Java. This document contains all the information that a client program needs to invoke the service, whether it is written in Java or another programming language. In the next section, we write a Java program that invokes the Amazon e-commerce service, using the WSDL provided by Amazon. We have no idea in which language that service was implemented.

Using JAX-WS

There are several toolkits for implementing web services in Java. In this section, we discuss the JAX-WS technology that is included in Java SE 6 and above.

With JAX-WS, you do not provide an interface for a web service. Instead, you annotate a class with @WebService, as shown in *Warehouse.java (Web service and JAX-WS)*. Note also the @WebParam annotation of the description parameter. It gives the parameter a humanly readable name in the WSDL file. (This annotation is optional. By default, the parameter would be called arg0.)

In RMI, the stub classes were generated dynamically, but with JAX-WS, you run a tool to generate them. Change to the base directory of the Webservices1 source and run the wsgen class as follows:

```
wsgen -classpath . com.horstmann.corejava.Warehouse
```

The tool generates two rather mundane classes in the *com.horstmann.corejava.jaxws* package. The first class encapsulates all parameters of the call:

```
public class GetPrice
{
    private String description;
    public String getDescription() { return this.description; }
    public void setDescription(String description) { this.description
= description; }
```

```
}
```

The second class encapsulates the return value:

```
public class GetPriceResponse
{
    private double _return;
    public double get_return() { return this._return; }
    public void set_return(double _return) { this._return = _return; }
}
```

Typically, one has a sophisticated server infrastructure for deploying web services, which we do not discuss here. The JDK contains a very simple mechanism for testing a service. Simply call the Endpoint.publish method. A server is started on the given URL—see *WarehouseServer.java (Web service and JAX-WS)*.

At this point, you should compile the server classes, run wsgen, and start the server:

```
java com.horstmann.corejava.WarehouseServer
```

Now point your web browser to *http://localhost:8080/WebServices/warehouse?wsdl*. You will get this WSDL file:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:tns="http://corejava.horstmann.com/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  targetNamespace="http://corejava.horstmann.com/"
name="WarehouseService">
  <types>
    <xsd:schema>
```

```
      <xsd:import
schemaLocation="http://localhost:8080/WebServices/warehouse?xsd=1"
      namespace="http://corejava.horstmann.com/"></xsd:import>
    </xsd:schema>
  </types>
  <message name="getPrice">
    <part element="tns:getPrice" name="parameters"></part>
  </message>
  <message name="getPriceResponse">
    <part element="tns:getPriceResponse" name="parameters"></part>
  </message>
  <portType name="Warehouse">
    <operation name="getPrice">
      <input message="tns:getPrice"></input>
      <output message="tns:getPriceResponse"></output>
    </operation>
  </portType>
  <binding name="WarehousePortBinding" type="tns:Warehouse">
    <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"></soap:binding>
    <operation name="getPrice">
      <soap:operation soapAction=""></soap:operation>
```

```
      <input><soap:body use="literal"></soap:body></input>
      <output><soap:body use="literal"></soap:body></output>
    </operation>
  </binding>
  <service name="WarehouseService">
    <port name="WarehousePort" binding="tns:WarehousePortBinding">
      <soap:address
location="http://localhost:8080/WebServices/warehouse"></soap:address>
    </port>
  </service>
</definitions>
```

This description tells us that an operation getPrice is provided. Its input is a *tns:getPrice* and itsoutput is a *tns:getPriceResponse*. (Here, tns is the namespace alias for the target namespace, http://corejava.horstmann.com.)

To understand these types, point your browser to *http://localhost:8080/WebServices/warehouse?xsd=1*. You will get this XSL document:

```
<xs:schema targetNamespace="http://corejava.horstmann.com/"
version="1.0">
   <xs:element name="getPrice" type="tns:getPrice"/>
   <xs:element name="getPriceResponse" type="tns:getPriceResponse"/>
   <xs:complexType name="getPrice">
      <xs:sequence><xs:element name="description" type="xs:string"
minOccurs="0"/></xs:sequence>
   </xs:complexType>
```

```
   <xs:complexType name="getPriceResponse">
      <xs:sequence><xs:element name="return"
type="xs:double"/></xs:sequence>
   </xs:complexType>
</xs:schema>
```

Now you can see that *getPrice* has a description element of type string, and *getPriceResponse* has a return element of type double.

A Web Service Client

Let's turn to implementing the client. Keep in mind that the client knows nothing about the server except what is contained in the WSDL. To generate Java classes that can communicate with the server, you generate a set of client classes, using the wsimport utility.

```
wsimport -keep -p com.horstmann.corejava.server
http://localhost:8080/WebServices/warehouse?wsdl
```

The -keep option keeps the source files, in case you want to look at them. The following classes and interfaces are generated:

```
GetPrice
GetPriceResponse
Warehouse
WarehouseService
ObjectFactory
```

You already saw the *GetPrice and GetPriceResponse* classes.

The Warehouse interface defines the remote getPrice method:

```
public interface Warehouse
```

```
{
    @WebMethod public double getPrice(@WebParam(name = "description")
String description);
}
```

You only need to know one thing about the *WarehouseService* class: its *getPort* method yields a stub of type Warehouse through which you invoke the service—see *WarehouseClient.java (web service and JAX-WS)*.

You can ignore the *ObjectFactory* class as well as the file *package-info.java* that defines a package-level annotation. (We discuss annotations in detail in Chapter 11.)

Now you are ready to run the client program. Double-check that the server is still running, open another shell window, and execute

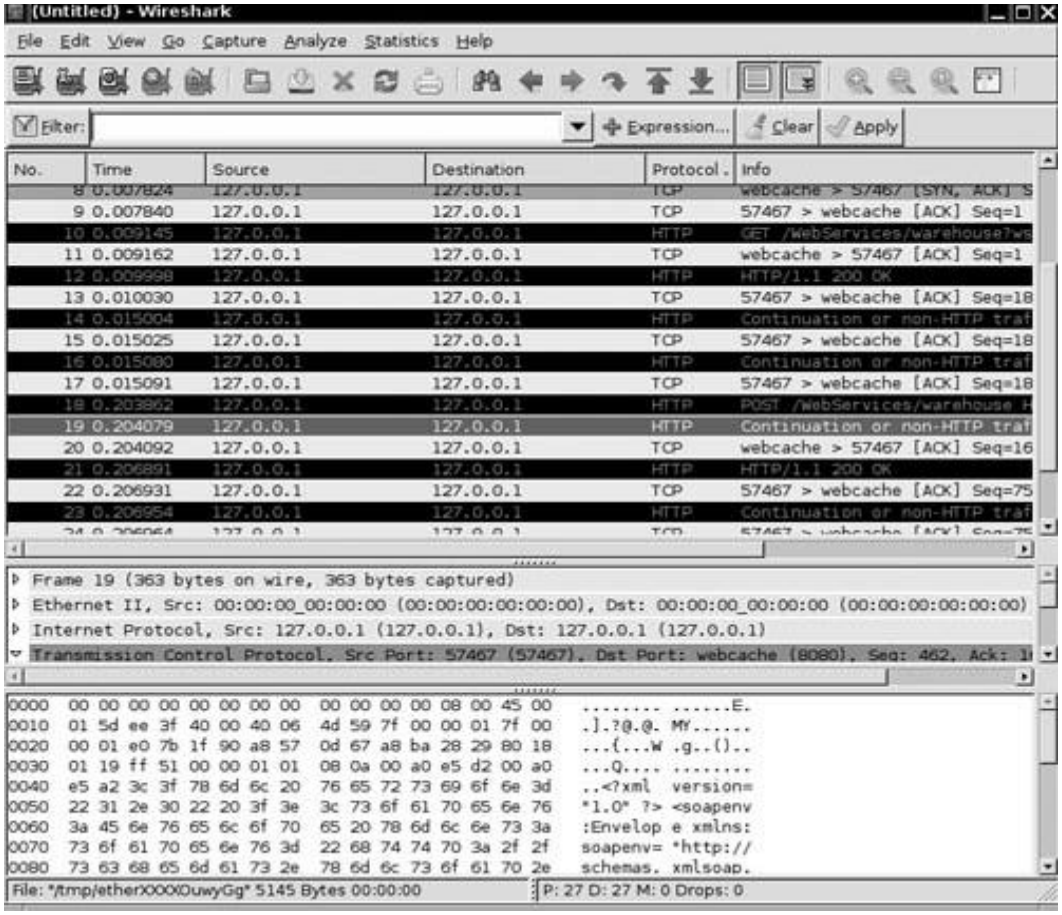
```
java WarehouseClient
```

You will get the familiar message about the price of a toaster.

We used a network sniffer to see how the client and server actually communicate (see Figure 44). The client sends the following request to the server:

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:ns1="http://corejava.horstmann.com/">
   <soapenv:Body>
      <ns1:getPrice><description>Blackwell
Toaster</description></ns1:getPrice>
   </soapenv:Body>
</soapenv:Envelope>
```

Figure 44. Analyzing SOAP traffic



The server responds:

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:ns1="http://corejava.horstmann.com/">
   <soapenv:Body>

<ns1:getPriceResponse><return>24.95</return></ns1:getPriceResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

In this section, you have seen the essentials about web services:

- The services are defined in a WSDL document, which is formatted as XML.
- The actual request and response methods use SOAP, another XML format.
- Clients and servers can be written in any language.

The Amazon E-Commerce Service

To make the discussion of web services more interesting, we look at a concrete example: the Amazon e-commerce web service, described at http://www.amazon.com/gp/aws/landing.html. The e-commerce web service allows a programmer to interact with the Amazon system for a wide variety of purposes. For example, you can get listings of all books with a given author or title, or you can fill shopping carts and place orders. Amazon makes this service available for use by companies that want to sell items to their customers, using the Amazon system as a fulfillment back end. To run our example program, you will need to sign up with Amazon and get a free developer token that lets you connect to the service.

Alternatively, you can adapt the technique described in this section to any other web service. The site http://www.xmethods.com lists many freely available web services that you can try.

Let us look more closely at the WSDL for the Amazon E-Commerce Service (located at http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl). It describes an *ItemSearch* operation as follows:

```
<operation name="ItemSearch">
   <input message="tns:ItemSearchRequestMsg"/>
   <output message="tns:ItemSearchResponseMsg"/>
</operation>
...
<message name="ItemSearchRequestMsg">
```

```
   <part name="body" element="tns:ItemSearch"/>
</message>
<message name="ItemSearchResponseMsg">
   <part name="body" element="tns:ItemSearchResponse"/>
</message>
```

Here are the definitions of the ItemSearch and ItemSearchResponse types:

```
<xs:element name="ItemSearch">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="MarketplaceDomain" type="xs:string"
minOccurs="0"/>
         <xs:element name="AWSAccessKeyId" type="xs:string"
minOccurs="0"/>
         <xs:element name="SubscriptionId" type="xs:string"
minOccurs="0"/>
         <xs:element name="AssociateTag" type="xs:string"
minOccurs="0"/>
         <xs:element name="XMLEscaping" type="xs:string"
minOccurs="0"/>
         <xs:element name="Validate" type="xs:string" minOccurs="0"/>
         <xs:element name="Shared" type="tns:ItemSearchRequest"
minOccurs="0"/>
         <xs:element name="Request" type="tns:ItemSearchRequest"
minOccurs="0"
            maxOcurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
```

```
</xs:element>

<xs:element name="ItemSearchResponse">

   <xs:complexType>

      <xs:sequence>

         <xs:element ref="tns:OperationRequest" minOccurs="0"/>

         <xs:element ref="tns:Items" minOccurs="0"
maxOccurs="unbounded"/>

      </xs:sequence>

   </xs:complexType>

</xs:element>
```

Using the JAX-WS technology, the ItemSearch operation becomes a method call:

```
void itemSearch(String marketPlaceDomain, String awsAccessKeyId,

   String subscriptionId, String associateTag, String xmlEscaping,
String validate,

   ItemSearchRequest shared, List<ItemSearchRequest> request,

   Holder<OperationRequest> opHolder, Holder<List<Items>>
responseHolder)
```

The *ItemSearchRequest* parameter type is defined as

```
<xs:complexType name="ItemSearchRequest">

   <xs:sequence>

      <xs:element name="Actor" type="xs:string" minOccurs="0"/>

      <xs:element name="Artist" type="xs:string" minOccurs="0"/>

      . . .

      <xs:element name="Author" type="xs:string" minOccurs="0"/>

      . . .
```

```
      <xs:element name="ResponseGroup" type="xs:string" minOccurs="0"
maxOccurs="unbounded"/>

      . . .

      <xs:element name="SearchIndex" type="xs:string" minOccurs="0"/>

      . . .

</xs:complexType>
```

This description is translated into a class.

```
public class ItemSearchRequest

{

   public ItemSearchRequest() { ... }

   public String getActor() { ... }

   public void setActor(String newValue) { ... }

   public String getArtist() { ... }

   public void setArtist(String newValue) { ... }

   ...

   public String getAuthor() { ... }

   public void setAuthor(String newValue) { ... }

   ...

   public List<String> getResponseGroup() { ... }

   ...

   public void setSearchIndex(String newValue) { ... }

   ...

}
```

To invoke the search service, construct an *ItemSearchRequest* object and call the itemSearch method of the "port" object.

```
ItemSearchRequest request = new ItemSearchRequest();

request.getResponseGroup().add("ItemAttributes");

request.setSearchIndex("Books");

Holder<List<Items>> responseHolder = new Holder<List<Items>>();

request.setAuthor(name);

port.itemSearch("", accessKey, "", "", "", "", request, null, null,
responseHolder);
```

The port object translates the Java object into a SOAP message, passes it to the Amazon server, translates the returned message into a *ItemSearchResponse* object, and places the response in the "holder" object.

Our sample application (in *AmazonTest.java*) is straightforward. The user specifies an author name and clicks the Search button. We simply show the first page of the response (see Figure 45). This shows that the web service is successful. We leave it as the proverbial exercise for the reader to extend the functionality of the application.

Figure 45. Connecting to a web service

To run this program, you first generate the client-side artifact classes:

```
wsimport -p com.horstmann.amazon
```

```
http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.
wsdl
```

Then edit the *AmazonTest.java* file to include your Amazon key, compile, and run:

```
javac AmazonTest.java

java AmazonTest
```

9.2 Demonstration of Distributed Objects

Warehouse.java

```
import java.rmi.*;
/**
The remote interface for a simple warehouse.
@version 1.0 2007-10-09
@author Cay Horstmann
*/
public interface Warehouse extends Remote
 {
    double getPrice(String description) throws RemoteException;
}
```

WarehouseImpl.java (for RMI programming part)

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
/**
* This class is the implementation for the remote Warehouse
interface.
* @version 1.0 2007-10-09
* @author Cay Horstmann
*/
public class WarehouseImpl extends UnicastRemoteObject implements
Warehouse
 {
public WarehouseImpl() throws RemoteException
```

```
    {
       prices = new HashMap<String, Double>();
       prices.put("Blackwell Toaster", 24.95);
       prices.put("ZapXpress Microwave Oven", 49.95);
    }
public double getPrice(String description) throws RemoteException
    {
       Double price = prices.get(description);
       return price == null ? 0 : price;
    }
    private Map<String, Double> prices;
}
```

WarehouseServer.java (for RMI programming part)

```
import java.rmi.*;
import javax.naming.*;
/**
* This server program instantiates a remote warehouse object,
registers it with the naming
* service, and waits for clients to invoke methods.
* @version 1.12 2007-10-09
* @author Cay Horstmann
 */
public class WarehouseServer
 {
public static void main(String[] args) throws RemoteException,
NamingException
    {
       System.out.println("Constructing server implementation...");
       WarehouseImpl centralWarehouse = new WarehouseImpl();
       System.out.println("Binding server implementation to
registry...");
       Context namingContext = new InitialContext();
       namingContext.bind("rmi:central_warehouse", centralWarehouse);
       System.out.println("Waiting for invocations from clients...");
   }
}
```

WarehouseClient.java

```
import java.rmi.*;
import java.util.*;
```

```
import javax.naming.*;
/**
* A client that invokes a remote method.
* @version 1.0 2007-10-09
* @author Cay Horstmann
*/
public class WarehouseClient
{
public static void main(String[] args) throws NamingException,
RemoteException
  {
        Context namingContext = new InitialContext();
        System.out.print("RMI registry bindings: ");
        Enumeration<NameClassPair> e =
namingContext.list("rmi://localhost/");
        while (e.hasMoreElements())
        System.out.println(e.nextElement().getName());
        String url = "rmi://localhost/central_warehouse";
        Warehouse centralWarehouse = (Warehouse)
namingContext.lookup(url);
        String descr = "Blackwell Toaster";
        double price = centralWarehouse.getPrice(descr);
        System.out.println(descr + ": " + price);
   }
}
```

javax.naming.InitialContext 1.3

- InitialContext()

constructs a naming context that can be used for accessing the RMI registry.

javax.naming.Context 1.3

- static Object lookup(String name)

returns the object for the given name. Throws a NamingException if the name is not currently bound.

- static void bind(String name, Object obj)

binds name to the object obj. Throws a NameAlreadyBoundException if the object is already bound.

- static void unbind(String name)

unbinds the name. It is legal to unbind a name that doesn't exist.

- static void rebind(String name, Object obj)

binds name to the object obj. Replaces any existing binding.

- NamingEnumeration<NameClassPair> list(String name)

returns an enumeration listing all matching bound objects. To list all RMI objects, call with "rmi:".

javax.naming.NameClassPair 1.3

- String getName()

gets the name of the named object.

- String getClassName()

gets the name of the class to which the named object belongs.

java.rmi.Naming 1.1

- static Remote lookup(String url)

returns the remote object for the URL. Throws a NotBoundException if the name is not currently bound.

- static void bind(String name, Remote obj)

binds name to the remote object obj. Throws an AlreadyBoundException if the object is already bound.

- static void unbind(String name)

unbinds the name. Throws the NotBound exception if the name is not currently bound.

- static void rebind(String name, Remote obj)

binds name to the remote object obj. Replaces any existing binding.

- static String[] list(String url)

returns an array of strings of the URLs in the registry located at the given URL. The array contains a snapshot of the names present in the registry.

Product.java

```
 import java.io.*;
public class Product implements Serializable
 {
public Product(String description, double price)
    {
       this.description = description;
       this.price = price;
   }
public String getDescription()
    {
       return description;
}
public double getPrice()
```

```
    {
       return price;
 }
public Warehouse getLocation()
    {
       return location;
}
public void setLocation(Warehouse location)
   {
       this.location = location;
  }
    private String description;
    private double price;
    private Warehouse location;
}
```

Book.java

```
 /**
* A book is a product with an ISBN number.
* @version 1.0 2007-10-09
* @author Cay Horstmann
*/
public class Book extends Product
 {
public Book(String title, String isbn, double price)
    {
       super(title, price);
       this.isbn = isbn;
   }
public String getDescription()
    {
       return super.getDescription() + " " + isbn;
   }
    private String isbn;
}
```

WarehouseImpl.java (parameter and return section)

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
 /**
* This class is the implementation for the remote Warehouse
```

```
interface.
* @version 1.0 2007-10-09
* @author Cay Horstmann
*/
public class WarehouseImpl extends UnicastRemoteObject implements
Warehouse
{
/**
* Constructs a warehouse implementation.
*/
public WarehouseImpl(Warehouse backup) throws RemoteException
    {
      products = new HashMap<String, Product>();
      this.backup = backup;
}
public void add(String keyword, Product product)
    {
       product.setLocation(this);
       products.put(keyword, product);
    }
public double getPrice(String description) throws RemoteException
    {
       for (Product p : products.values())
       if (p.getDescription().equals(description)) return
p.getPrice();
       if (backup == null) return 0;
       else return backup.getPrice(description);
    }
public Product getProduct(List<String> keywords) throws
RemoteException
    {
       for (String keyword : keywords)
       {
          Product p = products.get(keyword);
          if (p != null) return p;
       }
       if (backup != null)
          return backup.getProduct(keywords);
      else if (products.values().size() > 0)
          return products.values().iterator().next();
       else
       return null;
   }
    private Map<String, Product> products;
    private Warehouse backup;
```

```
}
```

WarehouseServer.java (parameter and return section)

```
import java.rmi.*;
import javax.naming.*;
/**
* This server program instantiates a remote warehouse objects,
registers it with the naming
* service, and waits for clients to invoke methods.
* @version 1.12 2007-10-09
* @author Cay Horstmann
 */
public class WarehouseServer
 {
public static void main(String[] args) throws RemoteException,
NamingException
    {
       System.setProperty("java.security.policy", "server.policy");
      System.setSecurityManager(new SecurityManager());
       System.out.println("Constructing server implementation...");
       WarehouseImpl backupWarehouse = new WarehouseImpl(null);
       WarehouseImpl centralWarehouse = new
WarehouseImpl(backupWarehouse);
       centralWarehouse.add("toaster", new Product("Blackwell
Toaster", 23.95));
       backupWarehouse.add("java", new Book("Core Java vol. 2",
"0132354799", 44.95));
       System.out.println("Binding server implementation to
registry...");
       Context namingContext = new InitialContext();
       namingContext.bind("rmi:central_warehouse", centralWarehouse);
       System.out.println("Waiting for invocations from clients...");
   }
}
```

WarehouseActivator.java

```
import java.io.*;
import java.rmi.*;
import java.rmi.activation.*;
import java.util.*;
import javax.naming.*;
```

```
/**
* This server program instantiates a remote warehouse object,
registers it with the naming
* service, and waits for clients to invoke methods.
* @version 1.12 2007-10-09
* @author Cay Horstmann
*/
public class WarehouseActivator
{
public static void main(String[] args) throws RemoteException,
NamingException,
         ActivationException, IOException
   {
       System.out.println("Constructing activation descriptors...");
       Properties props = new Properties();
       // use the server.policy file in the current directory
       props.put("java.security.policy", new
File("server.policy").getCanonicalPath());
       ActivationGroupDesc group = new ActivationGroupDesc(props,
null);
       ActivationGroupID id =
ActivationGroup.getSystem().registerGroup(group);
       Map<String, Double> prices = new HashMap<String, Double>();
       prices.put("Blackwell Toaster", 24.95);
       prices.put("ZapXpress Microwave Oven", 49.95);
MarshalledObject<Map<String, Double>> param = new
MarshalledObject<Map<String, Double>>(prices);
      String codebase = "http://localhost:8080/";
      ActivationDesc desc = new ActivationDesc(id, "WarehouseImpl",
codebase, param);
       Warehouse centralWarehouse = (Warehouse)
Activatable.register(desc);
      System.out.println("Binding activable implementation to
registry...");
      Context namingContext = new InitialContext();
       namingContext.bind("rmi:central_warehouse", centralWarehouse);
       System.out.println("Exiting...");
   }
}
```

WarehouseImpl.java (remote object activation)

```
import java.io.*;
import java.rmi.*;
```

```
import java.rmi.activation.*;
import java.util.*;
/**
* This class is the implementation for the remote Warehouse
interface.
* @version 1.0 2007-10-20
 * @author Cay Horstmann
 */
public class WarehouseImpl extends Activatable implements Warehouse
 {
public WarehouseImpl(ActivationID id, MarshalledObject<Map<String,
Double>> param)
      throws RemoteException, ClassNotFoundException, IOException
    {
       super(id, 0);
       prices = param.get();
       System.out.println("Warehouse implementation constructed.");
}
public double getPrice(String description) throws RemoteException
    {
      Double price = prices.get(description);
      return price == null ? 0 : price;
}
private Map<String, Double> prices;
}
```

rmid.policy

```
grant
{
     permission com.sun.rmi.rmid.ExecPermission
    "${java.home}${/}bin${/}java";
    permission com.sun.rmi.rmid.ExecOptionPermission
    "-Djava.security.policy=*";
}
```

java.rmi.activation.Activatable 1.2

- protected Activatable(ActivationID id, int port)

constructs the activatable object and establishes a listener on the given port. Use 0 for the port to have a port assigned automatically.

- static Remote exportObject(Remote obj, ActivationID id, int port)

makes a remote object activatable. Returns the activation receiver that should be made available to remote callers. Use 0 for the port to have a port assigned automatically.

- static Remote register(ActivationDesc desc)

registers the descriptor for an activatable object and prepares it for receiving remote calls. Returns the activation receiver that should be made available to remote callers.

java.rmi.MarshalledObject 1.2

- MarshalledObject(Object obj)

constructs an object containing the serialized data of a given object.

- Object get()

deserializes the stored object data and returns the object.

java.rmi.activation.ActivationGroupDesc 1.2

- ActivationGroupDesc(Properties props, ActivationGroupDesc.CommandEnvironment env)

constructs an activation group descriptor that specifies virtual machine properties for a virtual machine that hosts activated objects. The env parameter contains the path to the virtual machine executable and command-line options, or it is null if no special settings are required.

java.rmi.activation.ActivationGroup 1.2

- static ActivationSystem getSystem()

returns a reference to the activation system.

java.rmi.activation.ActivationSystem 1.2

- ActivationGroupID registerGroup(ActivationGroupDesc group)

registers an activation group and returns the group ID.

java.rmi.activation.ActivationDesc 1.2

- ActivationDesc(ActivationGroupID id, String className, String classFileURL, MarshalledObject data)

constructs an activation descriptor.

Warehouse.java (web service and JAX-WS)

```
package com.horstmann.corejava;
import java.util.*;
import javax.jws.*;
/**
* This class is the implementation for a Warehouse web service
```

```
* @version 1.0 2007-10-09
* @author Cay Horstmann
*/
@WebService
public class Warehouse
 {
public Warehouse()
    {
       prices = new HashMap<String, Double>();
       prices.put("Blackwell Toaster", 24.95);
       prices.put("ZapXpress Microwave Oven", 49.95);
}
public double getPrice(@WebParam(name="description") String
description)
    {
     Double price = prices.get(description);
     return price == null ? 0 : price;
    }
private Map<String, Double> prices;
}
```

WarehouseServer.java (web service and JAX-WS)

```
package com.horstmann.corejava;
import javax.xml.ws.*;
public class WarehouseServer
 {
public static void main(String[] args)
    {

Endpoint.publish("http://localhost:8080/WebServices/warehouse", new
Warehouse());
   }
}
```

WarehouseClient.java (web service and JAX-WS)

```
import java.rmi.*;
 import javax.naming.*;
 import com.horstmann.corejava.server.*;

 /**
 * The client for the warehouse program.
```

```
 * @version 1.0 2007-10-09
 * @author Cay Horstmann
 */
public class WarehouseClient
{
public static void main(String[] args) throws NamingException,
RemoteException
 {
     WarehouseService service = new WarehouseService();
    Warehouse port = service.getPort(Warehouse.class);
    String descr = "Blackwell Toaster";
    double price = port.getPrice(descr);
    System.out.println(descr + ": " + price);
}
}
```

AmazonTest.java

```
import com.horstmann.amazon.*;
import java.awt.*;
import java.awt.event.*;
import java.util.List;
import javax.swing.*;
import javax.xml.ws.*;
/**
* The client for the Amazon e-commerce test program.
* @version 1.10 2007-10-20
* @author Cay Horstmann
    /*
 public class AmazonTest
{
public static void main(String[] args)
{
    JFrame frame = new AmazonTestFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
/**
 * A frame to select the book author and to display the server
response.
 */
 class AmazonTestFrame extends JFrame
 {
   public AmazonTestFrame()
```

```
    {
      setTitle("AmazonTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      Panel panel = new JPanel();
       panel.add(new JLabel("Author:"));
       author = new JTextField(20);
       panel.add(author);
      JButton searchButton = new JButton("Search");
      panel.add(searchButton);
  searchButton.addActionListener(new ActionListener()
{
public void actionPerformed(ActionEvent event)
             {
               result.setText("Please wait...");
               new SwingWorker<Void, Void>()
                   {
                      @Override
protected Void doInBackground() throws Exception
                      {
                         String name = author.getText();
                        String books = searchByAuthor(name);
                         result.setText(books);
                         return null;
                      }
                  }.execute();
             }
     }   );
     result = new JTextArea();
    result.setLineWrap(true);
    result.setEditable(false);
   if (accessKey.equals("your key here"))
{
result.setText("You need to edit the Amazon access key.");
searchButton.setEnabled(false);
}
      add(panel, BorderLayout.NORTH);
     add(new JScrollPane(result), BorderLayout.CENTER);
  }
/**
* Calls the Amazon web service to find titles that match the author.
* @param name the author name
* @return a description of the matching titles
*/
private String searchByAuthor(String name)
{
```

```
     AWSECommerceService service = new AWSECommerceService();
     AWSECommerceServicePortType port =
service.getPort(AWSECommerceServicePortType.class);
     ItemSearchRequest request = new ItemSearchRequest();
     request.getResponseGroup().add("ItemAttributes");
request.setSearchIndex("Books");
  Holder<List<Items>> responseHolder = new Holder<List<Items>>();
        request.setAuthor(name);
        port.itemSearch("", accessKey, "", "", "", "", request, null,
null, responseHolder);
   List<Item> response = responseHolder.value.get(0).getItem();
        StringBuilder r = new StringBuilder();
       for (Item item : response)
      {
          r.append("authors=");
          List<String> authors =
item.getItemAttributes().getAuthor();
          r.append(authors);
          r.append(",title=");
          r.append(item.getItemAttributes().getTitle());
          r.append(",publisher=");
          r.append(item.getItemAttributes().getPublisher());
          r.append(",pubdate=");
          r.append(item.getItemAttributes().getPublicationDate());
          r.append("\n");
}
     return r.toString();
}
private static final int DEFAULT_WIDTH = 450;
private static final int DEFAULT_HEIGHT = 300;
private static final String accessKey = "12Y1EEATQ8DDYJCVQYR2";
private JTextField author;
private JTextArea result;
}
```

10 Scripting, Compiling, and Annotation Processing

This section discusses an overview of Scripting, Compiling, and Annotation Processing and the code demonstration.

10.1 Overview of Scripting, Compiling, and Annotation Processing

Scripting for the Java Platform

A scripting language is a language that avoids the usual edit/compile/link/run cycle by interpreting 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 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.

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. Figure 46 shows typical values.

Table 46. Properties of Scripting Engine Factories

Engine	Names	MIME types	Extensions
Rhino (included in Java SE 6)	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

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 6 includes a version of Rhino, a JavaScript interpreter developed by the Mozilla foundation. You can add additional 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://scripting.dev.java.net provides engines for a 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://scripting.dev.java.net).

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, then 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.

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 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("f.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 them 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. 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 need to implement 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.)

Redirecting Input and Output

You can redirect the standard input and output of a script by calling the *setReader* and *setWriter* method 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. 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.

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.

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 10.

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).

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.

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 that is 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, you only want to compile a script if you need to execute it repeatedly.

An Example: Scripting GUI Events

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

Have a look at the program in *ScriptTest.java*. *The ButtonFrame* class 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 properties file. Each property definition has the form

```
componentName.eventName = scriptCode
```

For example, if you choose to use JavaScript, you 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 that is 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 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://scripting.dev.java.net.

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 one 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.

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.

Compiling the Easy Way

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

```
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();

OutputStream outStream = ..., 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. Because the compiler takes no console input, you 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.

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, you 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 the file name, line number, and column number) as well as a human-readable description.

You obtain a *CompilationTask* object by calling 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 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, then you can ask the *StandardJavaFileManager* to translate 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, then you supply your own *JavaFileObject* subclass. Listing 11-2 shows the code for a source file object with data that are contained in a StringBuilder. The class extends the *SimpleJava FileObject* convenience class and overrides the *getCharContent* method to return the content of the string builder. We use this class in our example program in which we dynamically produce the code for a Java class and then compile it.

The *CompilationTask* class implements 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. *StringBuilderJavaSource.java* defines a class that implements the *JavaFileObject* interface. Its *openOutputStream* method returns the *ByteArrayOutputStream* into which the compiler will deposit the byte codes.

It turns out a bit tricky to tell the compiler's file manager to uses 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, you 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, you use the CompilationTask class instead. Its API is quite complex, but you can control every aspect of the compilation process.

An Example: Dynamic Java Code Generation

In 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. *ButtonFrame.java* shows a very simple example of a frame class, and *action.properties* 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 class. (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 *CompilerTest.java* 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 that are 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 byte codes 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 *MapClassLoader.java*) 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.

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 insert annotations into your code that your processing tool understands, and then apply the processing tool.

There is a wide range of uses for annotations, and that generality can be initially confusing. 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 start our discussion of annotations with the basic concepts and put them to use in a concrete example: We mark methods as event listeners for AWT components, and show you an annotation processor that analyzes the annotations and hooks up the listeners. We then discuss the syntax rules in detail. We 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 it 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 as @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 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 that is retained when the class file is loaded into the virtual machine. You have now seen the basic concepts of program metadata and annotations. In the next section, we walk through a concrete example of annotation processing.

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 design an annotation to avoid this drudgery. The annotation has the form

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

The programmer no longer has to make calls to addActionListener. Instead, each method is simply tagged with an annotation. *ButtonFrame.java (using annotations)* 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 *ActionListenerFor.java*

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 that 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 that is 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.

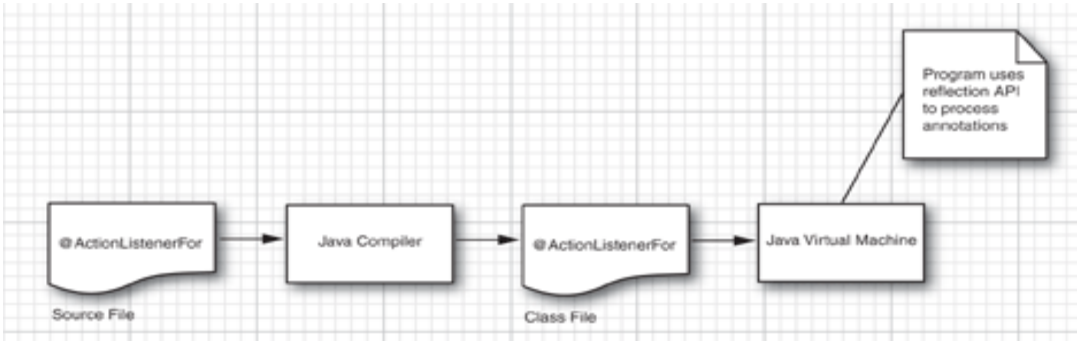
```
String fieldName = a.source();

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 that implements the *ActionListener* interface and 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 47 shows how annotations are handled in this example.

Figure 47. Processing annotations at runtime



In this example, the annotations were processed at runtime. It would also have been possible to process them at the source level. A source code generator might have produced the code for adding the listeners. Alternatively, the annotations might have been processed at the bytecode level. A bytecode editor might have injected 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 the section "Bytecode Engineering" on page 926.

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 easily refine the class to be 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.

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
{
   element declaration1
   element declaration2
   . . .
}
```

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]".

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, then the annotation *@BugReport(severity=10)* uses the new default, even in class files that have been compiled before the default changed.

If no elements are specified, either because the annotation doesn't have any or because all of them use the default value, then 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, then 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 we could have written the annotations as

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

instead of

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

All annotation interfaces implicitly extend the interface java.lang.annotation.Annotation. 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 for 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();

}
```

Because 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, . . .)
```

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, you enclose its values in braces, like this:

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

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

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

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

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

It is an error to introduce circular dependencies in annotations. For example, because

BugReport has an element of the annotation type Reference, then Reference can't 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.

You annotate a package 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, you can design an annotation that has a value of an array of simpler annotations:

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

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. Figure 48 shows these annotations. We discuss them in detail in the following two sections.

Figure 48. 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	An array of resources
`Generated`	All	Marks 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

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. 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 ISO8601 format) and a comment string are optional. For example,

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

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.

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
```

Figure 49 shows all possible values. They belong to the enumerated type *ElementType*. You can specify any number of element types, enclosed in braces.

Table 11-3. 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

In *ActionListenerFor.java*, 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.

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 50.

Figure 50. Documented annotations

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 in the same way as marker interfaces such as Serializable.

In fact, an annotation *@Serializable* would be more appropriate than the Serializable marker interfaces 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.

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 @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.

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. Java EE 5 uses 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; }
```

Property.java 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.

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 *BeanClassBeanInfo.java*. Declare the *BeanClassBeanInfo* 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 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.cla
ss)

      . . .

      return new java.beans.PropertyDescriptor[]

      {

         titlePositionDescriptor,

         . . .

      }

   }

}
```

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

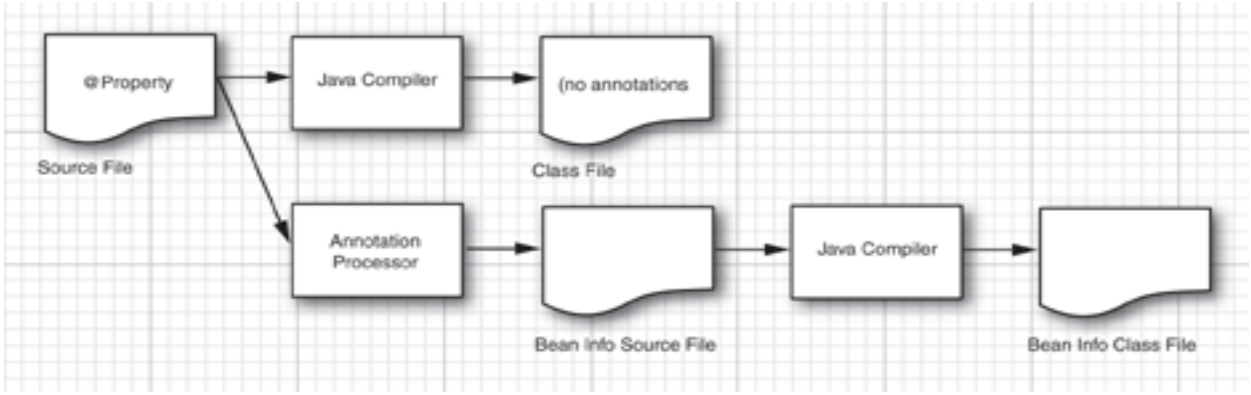
All this is easy enough to do, provided we can locate all methods that have been tagged with the *@Property* annotation.

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 ProcessorClassName1,ProcessorClassName2,...
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. If a processing round yields no further source files, then all source files are compiled. Figure 51 shows how the *@Property* annotations are processed.

Figure 51. Processing source-level annotations



We do not discuss the annotation processing API in detail, but the program in *BeanInfoAnnotationFactory.java* 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. Because the designers of the API love annotations, 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.

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 you with an annotated file, *ChartBean.java*. Compile the annotation processor:

```
javac BeanInfoAnnotationProcessor.java
```

Then run

```
javac -processor BeanInfoAnnotationProcessor
com/horstmann/corejava/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.

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.

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://java.sun.com/docs/books/vmspec). 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 task, 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:

   Code View:

```
ldc loggerName
invokestatic
java/util/logging/Logger.getLogger:(Ljava/lang/String;)Ljava/util/log
```

```
ging/Logger;

ldc className

ldc methodName

invokevirtual
java/util/logging/Logger.entering:(Ljava/lang/String;Ljava/lang/Strin
g;)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 *EntryLogger.java* edits a class file and inserts a logging call at the beginning of the methods that are annotated with the LogEntry annotation.

You need version 5.3 or later of the BCEL library to compile and run the EntryLogger program. (As this chapter was written, version 5.3 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 *Item.java*:

```
javac Item.java

javac -classpath .:bcel-version.jar EntryLogger.java

java -classpath .:bcel-version.jar EntryLogger Item
```

Try running

```
javap -c 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/logg
ing/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 *SetTest.java* 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

[[descripion=Toaster, partNumber=1729], [descripion=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. A bytecode editing tool picks up the directives and modifies the virtual machine instructions.

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 classloader 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 handle 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 that sets the Premain-Class attribute, for example:

   Premain-Class: *EntryLoggingAgent*

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

   ```
   javac -classpath .:bcel-version.jar EntryLoggingAgent
   ```

   ```
   jar cvfm EntryLoggingAgent.jar EntryLoggingAgent.mf 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=Item -classpath .:bcel-version.jar SetTest
```

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

EntryLoggingAgent.java 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. In other words, this technique carries out "just in time" modification of the bytecodes.

10.2 Demonstration of Scripting, Compiling, and Annotation Processing

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.

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.

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.

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 methods of the given object.

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.

ScriptTest.java

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

/**
* @version 1.00 2007-10-28
* @author Cay Horstmann
*/
public class ScriptTest
 {
public static void main(final String[] args)
    {
        EventQueue.invokeLater(new Runnable()
           {
 public void run()
              {
            String language;
                 if (args.length == 0) language = "js";
                 else language = args[0];

               ScriptEngineManager manager = new
ScriptEngineManager();
                System.out.println("Available factories: ");
               for (ScriptEngineFactory factory :
manager.getEngineFactories())
                  System.out.println(factory.getEngineName());
                final ScriptEngine engine =
manager.getEngineByName(language);

                if (engine == null)
                {
                   System.err.println("No engine for " + language);
                   System.exit(1);
                }
             ButtonFrame frame = new ButtonFrame();
                 try
                {
                   File initFile = new File("init." + language);
                   if (initFile.exists())
                   {
```

```
                        engine.eval(new FileReader(initFile));
                    }
                    getComponentBindings(frame, engine);
                     final Properties events = new Properties();
                    events.load(new FileReader(language +
".properties"));
                    for (final Object e : events.keySet())
                   {
                        String[] s = ((String) e).split("\\.");
                        addListener(s[0], s[1], (String)
events.get(e), engine);
                    }
                }
                catch (Exception e)
                {
                    e.printStackTrace();
                }

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setTitle("ScriptTest");
                frame.setVisible(true);
            }
        });
}
/**
* 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
*/
private static void addListener(String beanName, String eventName,
final String scriptCode,
final ScriptEngine engine) throws IllegalArgumentException,
IntrospectionException,
IllegalAccessException, InvocationTargetException
 {
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
{
for (EventSetDescriptor descriptor :
Introspector.getBeanInfo(bean.getClass())
.getEventSetDescriptors())
          if (descriptor.getName().equals(eventName)) return
descriptor;
return null;
}
}
class ButtonFrame extends JFrame
{
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);
}
public static final int DEFAULT_WIDTH = 300;
public static final int DEFAULT_HEIGHT = 200;
private JPanel panel;
private JButton yellowButton;
private JButton blueButton;
private JButton redButton;
}
```

StringBuilderJavaSource.java

```
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
{
/**
* 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');
}
private StringBuilder code;
}
```

ByteArrayJavaClass.java

```
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
 {
/**
* 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();
    }
    private ByteArrayOutputStream stream;
}
```

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.

- 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 produce the source code.

- OutputStream openOutputStream()

override this method for a file object that represents a class file, and produce a stream to which the byte codes 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.

ButtonFrame.java

```
package com.horstmann.corejava;
import javax.swing.*;

/**
 * @version 1.00 2007-11-02
 * @author Cay Horstmann
 */
```

```
public abstract class ButtonFrame extends JFrame
 {
 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();
 }

 protected abstract void addEventHandlers();

    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;
}
```

action.properties

```
1. yellowButton=panel.setBackground(java.awt.Color.YELLOW);
2. blueButton=panel.setBackground(java.awt.Color.BLUE);
3. redButton=panel.setBackground(java.awt.Color.RED);
```

CompilerTest.java

```
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 2007-11-02
 * @author Cay Horstmann
 */
public class CompilerTest
  {
 public static void main(final String[] args) throws IOException
     {
 JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();

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

 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);
              }
           };

JavaFileObject source =
buildSource("com.horstmann.corejava.ButtonFrame");
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<String, byte[]>();
                     for (ByteArrayJavaClass cl : classFileObjects)
                        byteCodeMap.put(cl.getName().substring(1),
cl.getBytes());
                     ClassLoader loader = new
MapClassLoader(byteCodeMap);
                    Class<?> cl = loader.loadClass("x.Frame");
                     JFrame frame = (JFrame) cl.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
      {
```

```
        StringBuilderJavaSource source = new
StringBuilderJavaSource("x.Frame");
        source.append("package x;\n");
        source.append("public class Frame extends " + superclassName
+ " {");
        source.append("protected void addEventHandlers() {");
        Properties props = new Properties();
        props.load(new FileReader("action.properties"));
        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;
    }
}
```

MapClassLoader.java

```
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
{
  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;
    }
     private Map<String, byte[]> classes;
}
```

ButtonFrame.java (using annotations)

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

/**
* A frame with a button panel
* @version 1.00 2004-08-17
* @author Cay Horstmann
*/
public class ButtonFrame extends JFrame
{
public ButtonFrame()
   {
      setTitle("ButtonTest");
      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);
    }

    public static final int DEFAULT_WIDTH = 300;
    public static final int DEFAULT_HEIGHT = 200;
    private JPanel panel;
    private JButton yellowButton;
    private JButton blueButton;
    private JButton redButton;
}
```

ActionListenerFor.java

```
 import java.lang.annotation.*;
/**
 * @version 1.00 2004-08-17
 * @author Cay Horstmann
 */
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ActionListenerFor
 {
   String source();
 }
```

ActionListenerInstaller.java

```
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 (Exception 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
*/
 public static void addListener(Object source, final Object param,
final Method m)         throws NoSuchMethodException,
IllegalAccessException, InvocationTargetException
```

```
   {
       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);
    }
 }
```

java.lang.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 that are present for this item, including inherited annotations. If no annotations are present, an array of length 0 is returned.

- Annotation[] getDeclaredAnnotations()

gets all annotations that are declared for this item, excluding inherited annotations. If no annotations are present, an array of length 0 is returned.

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 to another.

- int hashCode()

returns a hash code that is 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

Property.java

```
package com.horstmann.annotations;
 import java.lang.annotation.*;

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

BeanInfoAnnotationFactory.java

```
  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.*;
  import com.horstmann.annotations.*;

  /**
   * This class is the processor that analyzes Property annotations.
   * @version 1.10 2007-10-27
   * @author Cay Horstmann
   */

 @SupportedAnnotationTypes("com.horstmann.annotations.Property")
 @SupportedSourceVersion(SourceVersion.RELEASE_6)
 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,
Property>();
  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");
        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");
117.       }
       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();
}
}
```

EntryLogger.java

```
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
     {
/**
```

```
* 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
EntryLogger classname");
           else
              {
                JavaClass jc = Repository.lookupClass(args[0]);
         ClassGen cg = new ClassGen(jc);
         EntryLogger el = new EntryLogger(cg);
        el.convert();
        File f = new
File(Repository.lookupClassFile(cg.getClassName()).getPath());
                    cg.getJavaClass().dump(f.getPath());
      }
    }
    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("LLogEntry;"))
    {
        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();
```

```
      }
private ClassGen cg;
private ConstantPoolGen cpg;
}
```

Item.java

```
/**
* An item with a description and a part number.
* @version 1.00 2004-08-17
* @author Cay Horstmann
*/
public class Item
 {
 /**
* 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 description.equals(other.description) && partNumber ==
other.partNumber;
}
@LogEntry(logger = "global")
    public int hashCode()
{
      return 13 * description.hashCode() + 17 * partNumber;
}
private String description;
private int partNumber;
}
```

SetTest.java

```
import java.util.*;
import java.util.logging.*;
/**
* @version 1.01 2007-10-27
* @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<Item>();
        parts.add(new Item("Toaster", 1279));
        parts.add(new Item("Microwave", 4104));
        parts.add(new Item("Toaster", 1279));
        System.out.println(parts);
    }
}
```

EntryLoggingAgent.java

```
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;
                    }
                 }
            });
      }
}
```

11 Conclusions/Summary

This report goes through all concepts and code demonstrations from the basic to advance of all topics: Streams and Files, XML, Database Programming, Internationalizations, Advanced Swing, Advanced AWT, Javabeans Components, Distributed Objects, Scripting Compiling, Compiling and Annotation Processing. The report covers most of major parts of the topics. We withdrew a lot of lessonlearned from the project such as: understanding clearly the libraries before use and making use of them the most we can where appropriate. They're very powerful once we master it. It also helps to save lots of memories for our program.

References

[1] Core Java™ Volume II–Advanced Features, Eighth Edition, by Cay S. Horstmann & Gary Cornell, Prentice Hall, 2008

[2] http://java.sun.com/developer/onlineTraining/Programming/JDCBook/