File I/O

SOLUTION 1

Copying and Deleting Directories and Files

PROBLEM Your Java application needs to be able to copy and delete individual files and entire directories. You also want the option of copying and deleting nested file directories. You need to get lists of files in directories.

SOLUTION I will present a Java utility class *FileUtils* that you can use in your Java programs to copy and delete files and nested directories and also get directory listings.

One of the advantages of the Java platform is that once you know a technique like opening and reading files, that technique works on all computing platforms that support Java. In this solution you will build on the inherent portability of Java with a Java class library for general file manipulations that will solve most of your file handling problems on any platform. We will start by discussing the API of the class *`FileUtils`* in order to get you started quickly. Later, we will look at the implementation details.

It will be helpful if you open the file `FileUtils.java` in the directory `S01` in your favorite text editor or Java IDE while reading this section. This class has five static methods:

- The method `getNonDirFiles(String path)` returns all files in a specified directory path that are not subdirectory files:

```
static public Vector getNonDirFiles(String path)
     throws IOException;
```

- The method `getFiles(StringPath, String[] extensions)` returns all files in a directory path that end in file extensions that are passed to the method in an array of strings:

```
static public Vector getFiles(String path, String[] extensions)
     throws IOException;
```

- The method `getFiles(String path)` returns a vector of all files in the directory path. The elements of the returned vector are strings that include the full path to each file:

```
public static Vector getFiles(String path) throws IOException
```

- The method *`copyFilePath`* will copy either single files or entire directories, depending on if the first argument is a file path or a directory path name:

```
public static final void copyFilePath (File sourceFilePath,
                                       File destinationFilePath)
     throws IOException;
```

- The method *deleteFilePath* deletes both single files and recursively deletes entire directories:

```
public static final boolean deleteFilePath(File filePath);
```

All five methods are public static. Since this class has no instance data, the class never maintains state. It makes sense to make the methods static so that you do not need to create a class instance in order to use these utility methods. Here is an example for calling the first three methods for listings files:

```
Vector non_dir_files = FileUtils.getNonDirFiles("./SO1");
Vector txt_java_files = FileUtils.getFiles("./SO1",
       new String[]{".txt", ".java"});
Vector all_files = FileUtils.getFiles("./SO1");
```

In all cases, the returned vectors contain strings that are complete path names for the listed files. When calling `getFiles(StringPath, String[] extensions)`, you build an array of strings containing file extensions that you want to use for fetching files.

The method *copyFilePath* copies either a single file or a complete directory (including recursive copying of subdirectories), depending on whether the first argument, *sourceFilePath*, is a path to a file or a path to a directory. If the first argument is a path to an individual file, then the second argument, *destinationFilePath*, must also be a path to an individual file. If the first argument is a path to a directory, then the second argument, `destinationFilePath`, must also be a path to a directory. Here is an example of copying the contents of the directory `./temp123` to `./temp234` and then deleting the original directory `./temp123`:

```
FileUtils.copyFilePath(new File("./temp123"),
                       new File("./temp234"));
FileUtils.deleteFilePath(new File("./temp123"));
```

The method *deleteFilePath* takes a file argument that can be either an individual file or a directory.

Implementation Details

Using the `java.io` package classes, it is fairly simple get the contents of a file directory, as shown in the following example:

```
String path = "./SO1"; // must be a directory
File dir = new File(path);
// create a custom file filter to
// accept all non-directory files:
AllNonDirFileFilter filter = new AllNonDirFileFilter();
String[] ss = dir.list(filter);
```

The method `list()` either takes no arguments, in which case all files are returned in the specified directory, or it takes one argument that is an instance of a class implementing the `java.io.FilenameFilter` interface. Classes implementing this interface must define the following method:

```
public boolean accept (File dir, String name)
```

The file `FileUtil.java` includes three inner classes that all implement the *`FilenameFilter`* interface: *`LocalFileFilter`* (constructor takes an array of strings defining allowed file extensions), *`AllNonDirFileFilter`* (empty constructor, accepts all files that are not directory files), and *`AllDirFileFilter`* (empty constructor, accepts all files that are directory files).

Using the `java.io` package classes, it is simple to copy and delete single files, but there is no support for copying and deleting entire nested directories. The class *`FileUtils`* methods *`copyFilePath`* and *`deleteFilePath`* use recursion to process nested directories. The processing steps used by the method *`copyFilePath`* are

1. Check that both input arguments (source and target files) are either both single files or both file directories.
2. If you are copying directories, get all files in the source directory: for each file, if it is a single file, copy it; if it is a directory file, recursively call `copyFilePath` on the subdirectory.

The processing steps used by the method *`deleteFilePath`* are

1. If the input file path is a directory, get all files in the directory and recursively call deleteFilePath on each file.
2. If the input file path is a single file, delete it.

You can refer to the source file `FileUtils.java` for coding details.

SOLUTION 2

Reading and Writing JAR/ZIP Files

PROBLEM You want to be able to write data out in a compressed format (JAR, ZIP, and GZIP) and later read this data. You want to store different data entries in a single ZIP or JAR file.

SOLUTION You will use the standard utility classes in the packages `java.util.zip` (for both ZIP and GZIP support) and `java.util.jar` (for JAR support) to create and read compressed ZIP and JAR files and to create compressed GZIP streams.

The APIs for handling ZIP and JAR files are almost identical. In fact, the internal formats of ZIP and JAR files *are* identical except that a separate manifest file can be optionally added to a JAR file (actually, you could create a manifest file and add it to a ZIP file also).

I will cover examples of writing and reading ZIP files and JAR files; the JAR example program in directory *S02* is identical to the ZIP example program except I use *JarOutputStream* instead of *ZipOutputStream*, *JarInputStream* instead of *ZipInputStream*, *and JarEntry instead of ZipEntry*. The GNU ZIP format is popular on Unix, Linux, and Mac OS X; the classes *GZipOutputStream* and *GZipInputStream* can be used if you prefer the GNU ZIP format to compress a stream of data.

ZIP and JAR files contain zero or more entries. An entry contains the path of the original file and a stream of data for reading the original file's contents. The GZIP classes do not support archiving a collection of named data items like the ZIP and JAR utility classes. GZIP is used only for compressing and decompressing streams of data. The directory *S02* contains three sample programs:

ReadWriteZipFiles Demonstrates writing a ZIP file, closing it, and then reading it

ReadWriteJarFiles Demonstrates writing a JAR file, closing it, and then reading it

GZIPexample Demonstrates compressing a stream and then decompressing it

After you run the *ReadWriteZipFiles* example, the file `test.zip` will be created. Here is a list of this ZIP archive file:

```
markw% unzip -l test.zip
Archive:  test.zip
 Length     Date    Time    Name
 ------    ----    ----    ----
     66  05-18-03  11:31   test.txt
     19  05-18-03  11:31   string-entry
 ------                    -------
     85                    2 files
```

The entry named `test.txt` was created from the contents of the file `test.txt`. The entry named `string-entry` was created from the data in a Java string.

NOTE If you are extracting the files from a ZIP file containing nested directories, you will see entry names that include the full file path for the included files. If you are saving files stored in a ZIP archive, you need to create the full directory path for the files that you are extracting. *ZipEntry* objects that contain only a directory entry can be determined by using the method *isDirectory()*. You can create a new file directory for every directory *ZipEntry* object. Alternatively, you can discard the directory path information in the ZIP entry names and extract the files to the current directory.

After you run the *ReadWriteJarFiles* example, the file `test.jar` will be created. Here is a list of this JAR archive file:

```
markw% jar tvf test.jar
    66 Mon May 19 14:40:26 MST 2003 test.txt
    19 Mon May 19 14:40:26 MST 2003 string-entry
```

The entry named `test.txt` was created from the contents of the file `test.txt`. The entry named `string-entry` was created from the data in a Java string.

The *`GZIPexample`* program demonstrates compressing a file stream that is written to disk creating a file named `test.gz`.

NOTE The GZIP utility classes are also useful for compressing data streams transferred, for example, via network sockets (which are covered in Solutions 29 and 30). You will also look at Java streams in more detail in Solution 5 when you write a custom stream class.

Implementation Details

We will look at the ZIP example program in some detail; the JAR example program is identical except for the class substitutions mentioned earlier. Java allows I/O streams to be nested, so you can create a standard file output stream, then "wrap" it in a *`ZipOutputStream`*:

```
FileOutputStream fStream = new FileOutputStream("test.zip");
ZipOutputStream zStream = new ZipOutputStream(fStream);
```

The data that you write to a ZIP stream includes instances of the class *`ZipEntry`*:

```
ZipEntry stringEntry = new ZipEntry("string-entry");
zStream.putNextEntry(stringEntry);
String s = "This is\ntest data.\n";
zStream.write(s.getBytes(), 0, s.length());
zStream.closeEntry();  // close this ZIP entry
zStream.close();  // close the ZIP output stream
```

NOTE In this example, I simply wrote the contents of a string to the output stream associated with the ZIP entry; a more common use would be to open a local file, read the file's contents, and write the contents to the ZIP entry output stream.

The instance of class *`ZipEntry`* labels data written to the ZIP output stream. You can then write any data as bytes to ZIP output stream for this entry. After you close the ZIP entry, you can either add additional ZIP entries (and the data for those entries) or, as I do here, immediately close the ZIP output stream (which also closes the file output stream). The example program `ReadWriteZipFiles.java` writes two ZIP entries to the ZIP output stream.

To read a ZIP file, you first get a `java.util.Enumeration` object containing all the ZIP entries in the file:

```
ZipFile zipFile = new ZipFile("test.zip");
Enumeration entries = zipFile.entries();
```

The example program loops over this enumeration; here I will just show the code for reading a single ZIP entry:

```
ZipEntry entry = (ZipEntry) entries.nextElement();
InputStream is = zipFile.getInputStream(entry);
int count;
while (true) {
 count = is.read(buf);
 if (count < 1) break; // break: no more data is available
    // add your code here to process 'count' bytes of data:
}
is.close();
```

After reading each ZIP entry and processing the data in the entry, close the input stream for the entry.

The GZIP example program is very short, so I will list most of the code here:

```
// write a GZIP stream to a file:
String s = "test data\nfor GZIP demo.\n";
FileOutputStream fout = new FileOutputStream("test.gz");
GZIPOutputStream out = new GZIPOutputStream(fout);
out.write(s.getBytes());
out.close();
// read the data from a GZIP stream:
FileInputStream fin = new FileInputStream("test.gz");
GZIPInputStream in = new GZIPInputStream(fin);
byte[] bytes = new byte[2048];
while (true) {
     int count = in.read(bytes, 0, 2048);
     if (count < 1) break;
     String s2 = new String(bytes, 0, count);
     System.out.println(s2);
}
in.close();
```

You see again the utility of being able to wrap one input or output stream inside another stream. When you wrap a standard file output stream inside a GZIP output stream, you can still use the methods for the file output stream, but the data written will be compressed using the GZIP algorithm.

SOLUTION 3

Java Object Persistence with Serialization

PROBLEM You create Java objects in your programs that you would like to be able to save to disk and then later read them back into different programs.

SOLUTION You will use Java serialization to write objects to an object output stream and later read them back in from an object input stream.

The Java package `java.io` contains two stream classes that are able to respectively write and read Java objects that implement the `java.io.Serializable` interface: *`ObjectOutputStream`* and *`ObjectInputStream`*. Most Java standard library classes implement the *`Serializable`* interface, and it is simple to write your own Java classes so that they also implement the *`Serializable`* interface. The *`Serializable`* interface is a marker interface that contains no method signatures that require implementation: its purpose is to allow Java's persistence mechanism to determine which classes are intended to be serializable.

NOTE In addition to using Java object serialization as a lightweight and simple-to-use object persistence mechanism, it is also widely used in Remote Method Invocation (RMI), which I will cover in Solutions 30 and 31.

The file `SerializationTest.java` in the directory *`S03`* provides a complete example of defining a Java class that can be serialized, writing instances of this class and other standard Java classes to an output stream, then reading the objects back into the test program. While saving standard Java objects is simple, I will first cover the details of making your own Java classes implement the *`Serializable`* interface. Consider the nonpublic class *`DemoClass`* that is included at the bottom of the file `SerializationTest.java`:

```
class DemoClass implements Serializable {
    private String name = "";
    private transient Hashtable aHashtable = new Hashtable();

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public Hashtable getAHashtable() {return aHashtable; }
```

```
    public void setAHashtable(Hashtable aHashtable) {
        this.aHashtable = (Hashtable)aHashtable.clone();
    }
}
```

There are two aspects to this class definition that deal specifically with object serialization issues: the class implements the *Serializable* interface and the use of the keyword *transient* to prevent the serialization process from trying to save the hash table to the output stream during serialization and to prevent the deserialization process from trying to read input data to reconstruct the hash table. If you do not use the keyword *transient* when defining your Java classes, then all class variables are saved during serialization. All of a class's variables must also belong to classes that implement the *Serializable* interface or be marked with the *transient* keyword. The following code creates test data to be serialized:

```
// define some test data to be serialized:
Hashtable testHash = new Hashtable();
Vector testVector = new Vector();
testHash.put("cat", "dog");
testVector.add(new Float(3.14159f));
DemoClass demoClass = new DemoClass();
demoClass.setName("Mark");
Hashtable hash = demoClass.getAHashtable();
hash.put("cat", "dog");
```

The following code serializes this data:

```
// now, serialize the test data:
FileOutputStream ostream = new FileOutputStream("test.ser");
ObjectOutputStream p = new ObjectOutputStream(ostream);
p.writeObject(testHash);
p.writeObject(testVector);
p.writeObject(demoClass);
p.close();
```

The standard Java container classes `Hashtable` and `Vector` can be serialized if all the objects that they contain can also be serialized. If you try to serialize any object (including objects contained inside other objects) that cannot be serialized, then a `java.io.NotSerializableException` will be thrown. When you write the Java objects to a disk file `test.ser` in the test program, you'll once again wrap a file output stream inside another stream, this time an instance of class *ObjectOutputStream*. The class *ObjectOutputStream* adds an additional method that can be used on the output stream: *writeObject*.

It is also simple to read serialized objects from an input stream; in this case, I use a file input stream (wrapped in *ObjectInputStream*) to read the `test.ser` file:

```
InputStream ins = new java.io.FileInputStream("test.ser");
ObjectInputStream p = new ObjectInputStream(ins);
Hashtable h = (Hashtable)p.readObject();
```

The example program `SerializationTest.java` in the directory *S03* contains additional code demonstrating how to test the class type of objects as they are deserialized; here, you know that the first object that was written to the file was a hash table. Similarly, you can read the other two objects in the same order that they were written to the file `test.ser`:

```
Vector v = (Vector)p.readObject();
DemoClass demo = (DemoClass)p.readObject();
```

The object `demo` (class `DemoClass`) will not have the instance variable *aHashtable* defined. It has a null value because it was declared transient in the class definition.

NOTE No class constructor is called during deserialization. In the example program, the nonpublic class *DemoClass* has a default constructor (no arguments) that prints a message. When you run the example program, please note that this message is printed when an instance of *DemoClass* is constructed for serialization, but this message is not printed during deserialization.

Changing Java's Default Persistence Mechanism

The `SerializationTest.java` example program uses Java's default persistence mechanism with the modification of the class *DemoClass* using the *transient* keyword to avoid persisting one of the class variables. In general, you use the *transient* keyword for class data that either cannot be serialized or makes no sense to serialize (e.g., socket and thread objects). You can also customize Java's persistence mechanism. There are three ways to customize persistence:

- Implement custom protocol by defining the methods *writeObject* and *readObject*.
- Implement the *Externalizable* interface instead of the *Serializable* interface.
- Override default version control.

If you chose to customize the protocol for saving instances of one of your classes, then you must define two private methods:

```
private void writeObject(ObjectOutputStream outStream) throws IOException
private void readObject(ObjectInputStream inStream) throws IOException,
ClassNotFoundException
```

When you implement these methods, you will usually use two helper methods: *defaultWriteObject* in class *ObjectOutputStream* and *defaultReadObject* in class *ObjectInputStream*; for example:

```
private void readObject(ObjectInputStream inStream) throws IOException,
ClassNotFoundException {
    inStream.defaultReadObject();
    // place any custom initialization here
}
```

```
private void writeObject(ObjectOutputStream outStream) throws IOException {
    outStream.defaultWriteObject();
    // place any custom code here (not usually done)
}
```

You can alternatively create a new protocol for serializing your classes by not implementing the *Serializable* interface and instead implementing the *Externalizable* interface. To do this you must define two methods:

```
public void writeExternal(ObjectOutput out) throws IOException
public void readExternal(ObjectInput in)
    throws IOException, ClassNotFoundException
```

ObjectOutput is an interface that extends the *DataOutput* interface and adds methods for writing objects. *ObjectInput* is an interface that extends the *DataInput* interface and adds methods for reading objects.

NOTE Regardless of whether you use default serialization, write your own *readObject* and *writeObject* methods, or implement the *Externalizable* interface, you simply call the methods *readObject* and *writeObject* in your programs and the Java Virtual Machine (JVM) will automatically handle serialization correctly.

You might be wondering what happens if you change your class definitions after saving application objects to a disk file using serialization. Unfortunately, in the default case, you will not be able to read the serialized objects back because Java's default class version control mechanism will detect that the serialized objects were created from a different version of the class. However, there is a mechanism you can use to insure that serialized objects for a class are still valid for deserialization: overriding the default static class variable *serialVersionID*. For example, for the class `MyDemoClass`, use the Java toolkit utility program `serialver`:

```
> serialver MyDemoClass
MyDemoClass: static final long serialVersionID = 1274998123888721L
```

Then copy this generated code into your class definition:

```
public class MyDemoClass implements java.io.Serializable {
    static final long serialVersionID = 1274998123888721L;
    // … rest of class definition:
}
```

For example, if you now add a class variable, the deserialization process will continue without error, but the new variable will not have a defined value.

SOLUTION 4

Using Random Access File I/O

PROBLEM You need to randomly access data in a file.

SOLUTION Use the class *RandomAccessFile* to open a file for random access.

The class *RandomAccessFile* implements the *DataInput* and *DataOutput* interfaces, but it is not derived from the base classes *InputStream* and *OutputStream*. As a result, you cannot use the typical Java I/O stream functionality on instances of class *RandomAccessFile*. Think of a random access file as a randomly addressable array of bytes stored in a disk file. The file pointer acts like an index of this array of bytes. The class *RandomAccessFile* has three methods for accessing the file pointer of an open random access file:

void seek(long newFilePointer) Positions the file pointer to a new value.

int skipBytes(int bytesToSkip) Skips a specified number of bytes by moving the file pointer.

long getFilePointer() Returns the current value of the file pointer.

The example program `RandomAccessTest.java` in the directory `S04` opens a random access file for reading and writing and writes 50 bytes of data:

```
RandomAccessFile raf = new RandomAccessFile("test.random", "rw");
// define some data to write to the file:
byte[] data = new byte[50];
for (int i = 0; i < 50; i++) data[i] = (byte) i;
// by default, a random access file opens at the beginning,
// so just write the data:
raf.write(data, 0, 50);
```

The following code reads this data back in reverse order, which is inefficient but serves as an example. Here you are reading one byte of data at a time, starting at the end of the file and moving to the beginning of the file using the method *seek*:

```
// read back the data backwards (not efficient):
int count;
for (int i=49; i>=0; i--) {
```

```
        raf.seek(i);
        byte[] littleBuf = new byte[1];
        count = raf.read(littleBuf, 0, 1);
        if (count != 1) {
            System.out.println("error reading file at i="+i);
            break;
        }
        System.out.println("at i=" + i + ", byte read from file=" +
                           littleBuf[0]);
    }
    raf.close();
```

The constructors of class *RandomAccessFile* takes two arguments. The first argument can be either a string file path or an instance of class `java.io.File`. The second argument is always a string with one of these values:

"r" Opens file as read-only.

"rw" Opens file for reading and writing.

"rws" Opens the file for reading and writing and guarantees that both the file contents and file maintenance metadata are updated to disk before returning from any write operations.

"rwd" Opens the file for reading and writing and guarantees that the file contents are updated to disk before returning from any write operations.

The options `"rws"` and `"rwd"` are useful to guarantee that application-critical data is immediately saved to disk. There are several different methods for reading and writing byte data; in this example, I use the most general purpose read and write methods that take three arguments: an array of bytes, the starting index to use in the array of bytes, and the number of bytes to read or write. Although the logical model for random access files is an array of bytes, the class *RandomAccessFile* also has convenient methods for reading and writing Boolean, short, integer, long, and strings from a file, starting at the current file pointer position.

NOTE While there are legitimate uses for random access files, you should keep in mind the inefficiency of managing the storage of an array of bytes in a disk file. In some ways, random access files are a legacy from the times when random access memory (RAM) was expensive and it was important to reduce memory use whenever possible. For most Java applications, it is far more efficient to store data in memory and periodically serialize it to disk as required. A good alternative is to use a relational database to store persistent data using the JDBC portable database APIs (see Solutions 34–39).

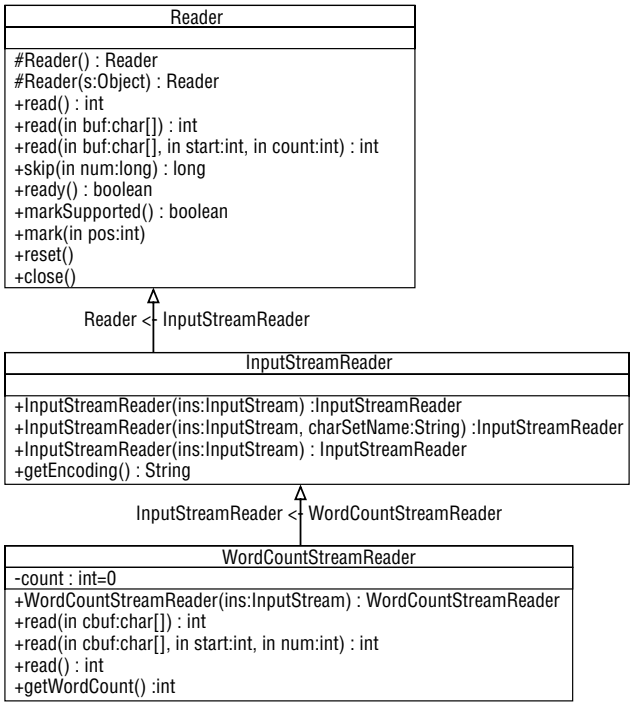
SOLUTION 5

Writing Text Filters

PROBLEM You need a framework for processing text files.

SOLUTION You will use a custom Java I/O stream class to perform text filtering by applying a sequence of filtering operations on input stream opened on the text data.

Before I get into implementing custom streams for supporting custom text filtering, we will take a brief overview of stream classes in the Java package `java.io`. The abstract base classes *`InputStream`* and *`OutputStream`* form the basis for byte handling streams. Streams for handling character data are derived from the base classes *`Reader`* and *`Writer`*. Since you are interested in processing text, in this solution we will only deal with character streams that are derived from the class *`Reader`*. The following illustration is a UML class diagram for the standard Java I/O classes *`Reader`* and *`InputStreamReader`*: the example class *`WordCountStreamReader`*, its base class *`InputStreamReader`*, and *`InputStreamReader's`* base class *`Reader`*.



Indirectly, the example class *WordCountStreamReader* inherits the methods of class *Reader* that are actually used to read from an input stream. You override these methods by calling the super class method and then performing application-specific filtering operations (in this case, counting the words in the character stream). In addition to defining a class constructor (which simply calls its super class constructor), the example class defines the method *getWordCount* that can be called at any time to get the number of space-delimited words read from the input stream.

The example stream class overrides all three read methods that are indirectly inherited from the class *Reader*. In each case, the read methods call the inherited read method with the same signature to actually read from the input stream, then look for word breaks:

```
public class WordCountStreamReader extends InputStreamReader {
    public WordCountStreamReader(InputStream ins) {
        super(ins);
    }
    public int read(char[] cbuf) throws IOException {
        int count =  super.read(cbuf);
        for (int i=0; i<count; i++) {
            if (cbuf[i] == ' ')  this.count++;
        }
        return count;
    }
    public int read(char[] cbuf, int start, int num)
                                          throws IOException {
        int count =  super.read(cbuf, start, num);
        for (int i=0; i<count; i++) {
            if (cbuf[i] == ' ')  this.count++;
        }
        return count;
    }
     public int read() throws IOException {
        int character =  super.read();
        if (character == ' ')  this.count++;
        return character;
    }
    public int getWordCount() {
        return count+1;
}
    private int count = 0;
}
```

NOTE In this example, you are simply counting spaces between words. In most real applications, you would use the `java.util.StringTokenizer` class to separate a string into individual words.

This is a simple class, but it demonstrates the basic strategy for defining new stream classes to act as text filters: use inherited methods to actually perform stream read operations while adding desired filtering behavior.

NOTE There are many applications that can use custom text filtering stream classes. For example, translating the input stream from English to German using the BabelFish web services, removing HTML markup from an HTML input stream (that is, leaving only plain text), and performing spelling correction.

The ability to wrap Java streams inside other Java streams is a powerful technique that is greatly enhanced by writing custom stream classes to perform application-specific filtering and information extraction operations.

TIP Remember: you can nest streams inside each other to any reasonable depth. For example, if you implemented a custom stream for translating English to German and another custom stream for stripping HTML tags, then your application might wrap (or nest) streams like this: `new WordCountStreamReader(new EnglishToGermanStreamReader (new StripHtmlTagsStreamReader(new InputStream(…))))`. This would allow you to count the translated German words without counting HTML tags.