CHAPTER

# 20 Input/Output: Exploring java.io

This chapter explores **java.io**, which provides support for I/O operations. Chapter 13 presented an overview of Java's I/O system, including basic techniques for reading and writing files, handling I/O exceptions, and closing a file. Here, we will examine the Java I/O system in greater detail.

As all programmers learn early on, most programs cannot accomplish their goals without accessing external data. Data is retrieved from an *input* source. The results of a program are sent to an *output* destination. In Java, these sources or destinations are defined very broadly. For example, a network connection, memory buffer, or disk file can be manipulated by the Java I/O classes. Although physically different, these devices are all handled by the same abstraction: the *stream*. An I/O stream, as explained in Chapter 13, is a logical entity that either produces or consumes information. An I/O stream is linked to a physical device by the Java I/O system. All I/O streams behave in the same manner, even if the actual physical devices they are linked to differ.

**NOTE** The stream-based I/O system packaged in **java.io** and described in this chapter has been part of Java since its original release and is widely used. However, beginning with version 1.4, a second I/O system was added to Java. It is called NIO (which was originally an acronym for New I/O). NIO is packaged in **java.nio** and its subpackages. The NIO system is described in Chapter 21.

**NOTE** It is important not to confuse the I/O streams used by the I/O system discussed here with the new stream API added by JDK 8. Although conceptually related, they are two different things. Therefore, when the term *stream* is used in this chapter, it refers to an I/O stream.

## The I/O Classes and Interfaces

The I/O classes defined by **java.io** are listed here:

| | | |
|---|---|---|
| BufferedInputStream | FileWriter | PipedOutputStream |
| BufferedOutputStream | FilterInputStream | PipedReader |

| BufferedReader | FilterOutputStream | PipedWriter |
|---|---|---|
| BufferedWriter | FilterReader | PrintStream |
| ByteArrayInputStream | FilterWriter | PrintWriter |
| ByteArrayOutputStream | InputStream | PushbackInputStream |
| CharArrayReader | InputStreamReader | PushbackReader |
| CharArrayWriter | LineNumberReader | RandomAccessFile |
| Console | ObjectInputStream | Reader |
| DataInputStream | ObjectInputStream.GetField | SequenceInputStream |
| DataOutputStream | ObjectOutputStream | SerializablePermission |
| File | ObjectOutputStream.PutField | StreamTokenizer |
| FileDescriptor | ObjectStreamClass | StringReader |
| FileInputStream | ObjectStreamField | StringWriter |
| FileOutputStream | OutputStream | Writer |
| FilePermission | OutputStreamWriter | |
| FileReader | PipedInputStream | |

The **java.io** package also contains two deprecated classes that are not shown in the preceding table: **LineNumberInputStream** and **StringBufferInputStream**. These classes should not be used for new code.

The following interfaces are defined by **java.io**:

| Closeable | FileFilter | ObjectInputValidation |
|---|---|---|
| DataInput | FilenameFilter | ObjectOutput |
| DataOutput | Flushable | ObjectStreamConstants |
| Externalizable | ObjectInput | Serializable |

As you can see, there are many classes and interfaces in the **java.io** package. These include byte and character streams, and object serialization (the storage and retrieval of objects). This chapter examines several commonly used I/O components. We begin our discussion with one of the most distinctive I/O classes: **File**.

## File

Although most of the classes defined by **java.io** operate on streams, the **File** class does not. It deals directly with files and the file system. That is, the **File** class does not specify how information is retrieved from or stored in files; it describes the properties of a file itself. A **File** object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies.

**NOTE** The **Path** interface and **Files** class, which are part of the NIO system, offer a powerful alternative to **File** in many cases. See Chapter 21 for details.

Files are a primary source and destination for data within many programs. Although there are severe restrictions on their use within applets for security reasons, files are still a central resource for storing persistent and shared information. A directory in Java is treated simply as a **File** with one additional property—a list of filenames that can be examined by the **list( )** method.

The following constructors can be used to create **File** objects:

File(String *directoryPath*)
File(String *directoryPath*, String *filename*)
File(File *dirObj*, String *filename*)
File(URI *uriObj*)

Here, *directoryPath* is the path name of the file; *filename* is the name of the file or subdirectory; *dirObj* is a **File** object that specifies a directory; and *uriObj* is a **URI** object that describes a file.

The following example creates three files: **f1**, **f2**, and **f3**. The first **File** object is constructed with a directory path as the only argument. The second includes two arguments—the path and the filename. The third includes the file path assigned to **f1** and a filename; **f3** refers to the same file as **f2**.

```
File f1 = new File("/");
File f2 = new File("/","autoexec.bat");
File f3 = new File(f1,"autoexec.bat");
```

**NOTE** Java does the right thing with path separators between UNIX and Windows conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly. Remember, if you are using the Windows convention of a backslash character (\), you will need to use its escape sequence (\\) within a string.

**File** defines many methods that obtain the standard properties of a **File** object. For example, **getName( )** returns the name of the file; **getParent( )** returns the name of the parent directory; and **exists( )** returns **true** if the file exists, **false** if it does not. The following example demonstrates several of the **File** methods. It assumes that a directory called **java** exists off the root directory and that it contains a file called **COPYRIGHT**.

```
// Demonstrate File.
import java.io.File;

class FileDemo {
  static void p(String s) {
    System.out.println(s);
  }

  public static void main(String args[]) {
    File f1 = new File("/java/COPYRIGHT");

    p("File Name: " + f1.getName());
    p("Path: " + f1.getPath());
    p("Abs Path: " + f1.getAbsolutePath());
    p("Parent: " + f1.getParent());
    p(f1.exists() ? "exists" : "does not exist");
```

```
    p(f1.canWrite() ? "is writeable" : "is not writeable");
    p(f1.canRead() ? "is readable" : "is not readable");
    p("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
    p(f1.isFile() ? "is normal file" : "might be a named pipe");
    p(f1.isAbsolute() ? "is absolute" : "is not absolute");
    p("File last modified: " + f1.lastModified());
    p("File size: " + f1.length() + " Bytes");
  }
}
```

This program will produce output similar to this:

```
File Name: COPYRIGHT
Path: \java\COPYRIGHT
Abs Path: C:\java\COPYRIGHT
Parent: \java
exists
is writeable
is readable
is not a directory
is normal file
is not absolute
File last modified: 1282832030047
File size: 695 Bytes
```

Most of the **File** methods are self-explanatory. **isFile( )** and **isAbsolute( )** are not. **isFile( )** returns **true** if called on a file and **false** if called on a directory. Also, **isFile( )** returns **false** for some special files, such as device drivers and named pipes, so this method can be used to make sure the file will behave as a file. The **isAbsolute( )** method returns **true** if the file has an absolute path and **false** if its path is relative.

**File** includes two useful utility methods of special interest. The first is **renameTo( )**, shown here:

boolean renameTo(File *newName*)

Here, the filename specified by *newName* becomes the new name of the invoking **File** object. It will return **true** upon success and **false** if the file cannot be renamed (if you attempt to rename a file so that it uses an existing filename, for example).

The second utility method is **delete( )**, which deletes the disk file represented by the path of the invoking **File** object. It is shown here:

boolean delete( )

You can also use **delete( )** to delete a directory if the directory is empty. **delete( )** returns **true** if it deletes the file and **false** if the file cannot be removed.

Here are some other **File** methods that you will find helpful:

| Method | Description |
|---|---|
| void deleteOnExit( ) | Removes the file associated with the invoking object when the Java Virtual Machine terminates. |
| long getFreeSpace( ) | Returns the number of free bytes of storage available on the partition associated with the invoking object. |
| long getTotalSpace( ) | Returns the storage capacity of the partition associated with the invoking object. |
| long getUsableSpace( ) | Returns the number of usable free bytes of storage available on the partition associated with the invoking object. |
| boolean isHidden( ) | Returns **true** if the invoking file is hidden. Returns **false** otherwise. |
| boolean setLastModified(long *millisec*) | Sets the time stamp on the invoking file to that specified by *millisec*, which is the number of milliseconds from January 1, 1970, Coordinated Universal Time (UTC). |
| boolean setReadOnly( ) | Sets the invoking file to read-only. |

Methods also exist to mark files as readable, writable, and executable. Because **File** implements the **Comparable** interface, the method **compareTo( )** is also supported.

JDK 7 added a method to **File** called **toPath( )**, which is shown here:

```
Path toPath( )
```

**toPath( )** returns a **Path** object that represents the file encapsulated by the invoking **File** object. (In other words, **toPath( )** converts a **File** into a **Path**.) **Path** is packaged in **java.nio.file** and is part of NIO. Thus, **toPath( )** forms a bridge between the older **File** class and the newer **Path** interface. (See Chapter 21 for a discussion of **Path**.)

## Directories

A directory is a **File** that contains a list of other files and directories. When you create a **File** object that is a directory, the **isDirectory( )** method will return **true**. In this case, you can call **list( )** on that object to extract the list of other files and directories inside. It has two forms. The first is shown here:

```
String[ ] list( )
```

The list of files is returned in an array of **String** objects.

The program shown here illustrates how to use **list( )** to examine the contents of a directory:

```
// Using directories.
import java.io.File;

class DirList {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);
```

```
    if (f1.isDirectory()) {
      System.out.println("Directory of " + dirname);
      String s[] = f1.list();

      for (int i=0; i < s.length; i++) {
        File f = new File(dirname + "/" + s[i]);
        if (f.isDirectory()) {
          System.out.println(s[i] + " is a directory");
        } else {
          System.out.println(s[i] + " is a file");
        }
      }
    } else {
      System.out.println(dirname + " is not a directory");
    }
  }
}
```

Here is sample output from the program. (Of course, the output you see will be different, based on what is in the directory.)

```
Directory of /java
bin is a directory
lib is a directory
demo is a directory
COPYRIGHT is a file
README is a file
index.html is a file
include is a directory
src.zip is a file
src is a directory
```

## Using FilenameFilter

You will often want to limit the number of files returned by the **list( )** method to include only those files that match a certain filename pattern, or *filter*. To do this, you must use a second form of **list( )**, shown here:

String[ ] list(FilenameFilter *FFObj*)

In this form, *FFObj* is an object of a class that implements the **FilenameFilter** interface.

**FilenameFilter** defines only a single method, **accept( )**, which is called once for each file in a list. Its general form is given here:

boolean accept(File *directory*, String *filename*)

The **accept( )** method returns **true** for files in the directory specified by *directory* that should be included in the list (that is, those that match the *filename* argument) and returns **false** for those files that should be excluded.

The **OnlyExt** class, shown next, implements **FilenameFilter**. It will be used to modify the preceding program to restrict the visibility of the filenames returned by **list( )** to files with names that end in the file extension specified when the object is constructed.

```
import java.io.*;

public class OnlyExt implements FilenameFilter {
  String ext;

  public OnlyExt(String ext) {
    this.ext = "." + ext;
  }

  public boolean accept(File dir, String name) {
    return name.endsWith(ext);
  }
}
```

The modified directory listing program is shown here. Now it will only display files that use the **.html** extension.

```
// Directory of .HTML files.
import java.io.*;

class DirListOnly {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);
    FilenameFilter only = new OnlyExt("html");
    String s[] = f1.list(only);

    for (int i=0; i < s.length; i++) {
      System.out.println(s[i]);
    }
  }
}
```

## The listFiles( ) Alternative

There is a variation to the **list( )** method, called **listFiles( )**, which you might find useful. The signatures for **listFiles( )** are shown here:

File[ ] listFiles( )
File[ ] listFiles(FilenameFilter *FFObj*)
File[ ] listFiles(FileFilter *FObj*)

These methods return the file list as an array of **File** objects instead of strings. The first method returns all files, and the second returns those files that satisfy the specified **FilenameFilter**. Aside from returning an array of **File** objects, these two versions of **listFiles( )** work like their equivalent **list( )** methods.

The third version of **listFiles( )** returns those files with path names that satisfy the specified **FileFilter**. **FileFilter** defines only a single method, **accept( )**, which is called once for each file in a list. Its general form is given here:

boolean accept(File *path*)

The **accept( )** method returns **true** for files that should be included in the list (that is, those that match the *path* argument) and **false** for those that should be excluded.

### Creating Directories

Another two useful **File** utility methods are **mkdir( )** and **mkdirs( )**. The **mkdir( )** method creates a directory, returning **true** on success and **false** on failure. Failure can occur for various reasons, such as the path specified in the **File** object already exists, or the directory cannot be created because the entire path does not exist yet. To create a directory for which no path exists, use the **mkdirs( )** method. It creates both a directory and all the parents of the directory.

## The AutoCloseable, Closeable, and Flushable Interfaces

There are three interfaces that are quite important to the stream classes. Two are **Closeable** and **Flushable**. They are defined in **java.io** and were added by JDK 5. The third, **AutoCloseable**, was added by JDK 7. It is packaged in **java.lang**.

**AutoCloseable** provides support for the **try**-with-resources statement, which automates the process of closing a resource. (See Chapter 13.) Only objects of classes that implement **AutoCloseable** can be managed by **try**-with-resources. **AutoCloseable** is discussed in Chapter 17, but it is reviewed here for convenience. The **AutoCloseable** interface defines only the **close( )** method:

void close( ) throws Exception

This method closes the invoking object, releasing any resources that it may hold. It is called automatically at the end of a **try**-with-resources statement, thus eliminating the need to explicitly call **close( )**. Because this interface is implemented by all of the I/O classes that open a stream, all such streams can be automatically closed by a **try**-with-resources statement. Automatically closing a stream ensures that it is properly closed when it is no longer needed, thus preventing memory leaks and other problems.

The **Closeable** interface also defines the **close( )** method. Objects of a class that implement **Closeable** can be closed. Beginning with JDK 7, **Closeable** extends **AutoCloseable**. Therefore, any class that implements **Closeable** also implements **AutoCloseable**.

Objects of a class that implements **Flushable** can force buffered output to be written to the stream to which the object is attached. It defines the **flush( )** method, shown here:

void flush( ) throws IOException

Flushing a stream typically causes buffered output to be physically written to the underlying device. This interface is implemented by all of the I/O classes that write to a stream.