**The I/O Classes and Interfaces**

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

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

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.

Files are a primary source and destination for data within many programs. Although there are severe restrictions on their use within untrusted code 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.

Example:

***File f = new File("abc.txt ");***

The above line won’t create any physical file first it will check is there any physical file named with abc.txt is available or not. If it is available f simply refers to that file if it is not available then we are just creating File Object to represent the name “abc.txt”.

We can use the java file object to represent the directory also

File f = new File(“nadeem”);

f.exists() // false

f.mkdir() // Creates a directory with name Nadeem in the current working directory

f.exists() // true

In UNIX everything is treated as a file, Java File IO concept is implemented based on UNIX operating system. Java File Object t can be used to represent files and directories.

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).

**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**.

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.

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. |

A method of special interest is 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.

**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.

//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 a 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);

}

}

// 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 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**. The third, **AutoCloseable**, 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 18, 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. **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.

**IO Exception**

Two exceptions play an important role in I/O handling. The first is **IOException**. As it relates to most of the I/O classes described in this chapter, if an I/O error occurs, an **IOException** is thrown. In many cases, if a file cannot be opened, a **FileNotFoundException** is thrown. **FileNotFoundException** is a subclass of **IOException**, so both can be caught with a single **catch** that catches **IOException**. For brevity, this is the approach used by most of the sample code in this chapter. However, in your own applications, you might find it useful to **catch** each exception separately.

Another exception class that is sometimes important when performing I/O is **SecurityException**. As explained in Chapter 13, in situations in which a security manager is present, several of the file classes will throw a **SecurityException** if a security violation occurs when attempting to open a file.

**Two Ways to Close a Stream**

In general, a stream must be closed when it is no longer needed. Failure to do so can lead to memory leaks and resource starvation. The techniques used to close a stream were described in Chapter 13, but because of their importance, they warrant a brief review here before the stream classes are examined.

There are two basic ways in which you can close a stream. The first is to explicitly call **close( )** on the stream. This is the traditional approach that has been used since the original release of Java. With this approach, **close( )** is typically called within a **finally** block. Thus, a simplified skeleton for the traditional approach is shown here:

try {

// open and access the file

} catch( I/O-exception) {

// ...

} finally {

// close the file

}

This general technique (or variation thereof) is common in code that predates JDK 7.

The second approach to closing a stream is to automate the process by using the **try**-with-resources statement that was added by JDK 7. The **try**-with-resources statement is an enhanced form of **try** that has the following form:

try (resource-specification) {

// use the resource

}

Typically, *resource-specification* is a statement or statements that declares and initializes a resource, such as a file or other stream-related resource. It consists of a variable declaration in which the variable is initialized with a reference to the object being managed. When the **try** block ends, the resource is automatically released. In the case of a file, this means that the file is automatically closed. Thus, there is no need to call **close( )** explicitly.

Here are three key points about the **try**-with-resources statement:

* Resources managed by **try**-with-resources must be objects of classes that implement **AutoCloseable**.
* A resource declared in the **try** is implicitly **final**. A resource declared outside the **try** must be effectively **final**.
* You can manage more than one resource by separating each declaration by a semicolon.