Input/Output Streams

(Programming III)

Streams

- Java programs perform I/O through streams.
- A *stream* is an abstraction that produces or consumes information.
- A stream is linked to a physical device by the Java I/O system.
- All streams behave in the same manner regardless of physical devices to which they are linked.

The same I/O classes and methods can be applied to different types of devices.

Byte Streams and Character Streams

- Java defines two types of streams:
  byte and ***character***.
- ***Byte streams*** provide a convenient means for handling input and output of bytes.
- Byte streams are used, for example, when reading or writing *binary data*.
- ***Character streams*** provide a convenient means for handling input and output of *characters*.
- They use *Unicode* and can be internationalised.
- In some cases, character streams are *more efficient* than byte streams.
- At the lowest level, *all* I/O is still *byte-oriented*.
-

Byte Stream Classes

At the top are two *abstract* classes:
InputStream
OutputStream
Each of these abstract classes has several concrete subclasses that handle the differences among various devices, such as
disk files,
network connections, and
memory buffers.
To use stream classes, we must import `java.io`

Stream Class	**Meaning**
`BufferedInputStream`	Buffered input stream
`BufferedOutputStream`	Buffered output stream
`ByteArrayInputStream`	Input stream that reads from a byte array
`ByteArrayOutputStream`	Output stream that writes to a byte array

`DataInputStream`	An input stream that contains methods for reading the Java standard data types
`DataOutputStream`	An output stream that contains methods for writing the Java

	standard data types
`FileInputStream`	Input stream that reads from a file
`FileOutputStream`	Output stream that writes to a file
`FilterInputStream`	Implements `InputStream`
`FilterOutputStream`	Implements `OutputStream`
`InputStream`	Abstract class that describes stream input
`ObjectInputStream`	Input stream for objects
`ObjectOutputStream`	Output stream for objects
`OutputStream`	Abstract class that describes stream output
`PipedInputStream`	Input pipe
`PipedOutputStream`	Output pipe

`PrintStream`	Output stream that contains `print()` and `println()`
`PushbackInputStream`	Input stream that supports one-byte "unget," which returns a byte to the input stream

`SequenceInputStream`	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

Character Stream Classes

At the top are two *abstract* classes:

Reader and

Writer.

Handle Unicode character streams.

Java has several concrete subclasses of each of these.

To use Character stream class `java.io` package should be imported

Stream Class	**Meaning**
`BufferedReader`	Buffered input character stream
`BufferedWriter`	Buffered output character stream
`CharacterArrayReader`	Input stream that reads from a character array
`CharacterArrayWriter`	Output stream that writes to a character array
`FileReader`	Input stream that reads from a file
`FileWriter`	Output stream that writes to a file
`FilterReader`	Filtered reader
`FilterWriter`	Filtered writer
`InputStreamReader`	Input stream that translates bytes to characters

`LineNumberReader`	Input stream that counts lines
`OutputStreamWriter`	Output stream that translates characters to bytes
`PipedReader`	Input pipe
`PipedWriter`	Output pipe
`PrintWriter`	Output streams that contain `print()` and `println()`
`PushbackReader`	Input stream that allows characters to be returned to the input stream

`Reader`	Abstract class that describes character stream input
`StringReader`	Input stream that reads from a string
`StringWriter`	Output stream that writes to a string
`Writer`	Abstract class that describes character stream output

Note: The abstract classes `Reader` and `Writer` define several key methods including
`read()` and
`write()`.

Predefined Streams

A class called `System`, defined in the package `java.lang` contains three predefined streams variables: `in`, `out`, and `err`, and declared as `public`, `static` and `final`
`System.out` refers to the standard output stream (the console, by default).
`System.in` refers to standard input (keyboard, by default)
`System.err` refers to the standard error stream, (console by default).
These streams may be redirected to any compatible I/O device in addition to default devices.

`System.in` is an object of type `InputStream`;
`System.out` and `System.err` are objects of type `PrintStream`.

These are *byte streams* though typically used to read and write characters.

Reading Console Input

- Console input is accomplished by reading from `System.in`.
- To obtain a character-based stream that is attached to the console, `System.in` is wrapped in a `BufferedReader` object.
- `BufferedReader` supports a buffered input stream.

A commonly used constructor is shown here: `BufferedReader(Reader inputReader)`
Here the inputReader is the stream that is linked to the instance of `BufferedReader` that is being created.

`System.in` refers to an object type of `InputStream` and can be used for *inputStream*

`InputStreamReader isr = new InputStreamReader(System.in)`

isr can be used to obtain a buffered reader that is connected to the keyboard as follows:

```
BufferedReader br = new BufferedReader(isr)
```

equivalently,

```
BufferedReader br =
new BufferedReader(new InputStreamReader(System.in))
```

`br` is a character-based stream that is linked to console through `System.in`

Reading Characters

A version of `read()` can be used to read characters from a `BufferedReader`

```
 int read() throws IOException
```

`read()` reads a character from the input stream and returns it as an integer.

Returns -1, when the end of stream is encountered

```
// Use a BufferedReader to read characters from the console.
import java.io.*; class BRRead {
  public static void main(String
        args[]) throws IOException
    { char c;
      BufferedReader br = new
      BufferedReader(new
   InputStreamReader(System.in));
      System.out.println("Enter characters," +
                          " 'q' to quit.");
     // read characters
      do {
           c = (char) br.read();
           System.out.println(c);
         } while(c != 'q');
    }//main
}//class
```

Sample Input

```
Queue1-2quench
```

Output

```
Q
u
e
u
e
1
-
2
q
```

Reading Strings

A version of `readln()` belonging to BufferedReader can be used to read string from the keyboard.

```
String readLine() throws IOException
```

Buffered Read Lines

```
// Read a string from console using a
//  BufferedReader.
import java.io.*;
class BRReadLines {

     public static void main(String
               args[]) throws IOException
     {
          // create a BufferedReader using System.in

          BufferedReader br = new
                    BufferedReader(new
                    InputStreamReader(System.in));

          String str;

          System.out.println("Enter lines of text.");

          System.out.println("Enter 'stop' to quit.");
          do {

               str = br.readLine();

               System.out.println(str);

          } while(str.equals("stop")); //”stop”.equals(str)
     }
}
```

Tiny Edit

```
// A tiny editor.
import java.io.*;
class TinyEdit {
     public static void main(String args[]) throws IOException
     {
          // create a BufferedReader using System.in
          BufferedReader br = new BufferedReader(new
                    InputStreamReader(System.in));
          String str[] = new String[100];
          System.out.println("Enter lines of text.");
          System.out.println("Enter 'stop' to quit.");
          for(int i=0; i<100; i++) {
               str[i] = br.readLine();
               if(str[i].equals("stop")) break;

          }
          System.out.println("\nHere is your file:");
          // display the lines
          for(int i=0; i<100; i++) {
               if(str[i].equals("stop")) break;
               System.out.println(str[i]);
          }
     }
}
```

Writing Console Output

The methods `print( )` and `println( )` can be used for console output.
These methods are defined by the class `PrintStream` (which is the type of object referenced by `System.out`).

`System.out` is a *byte stream*, used for simple program output.

`PrintStream` is an output stream derived from `OutputStream`, and it implements the low-level `method write( )`.
`write( )` can be used to write to the console.

`void write(int byteval)` Although byteval is declared as an integer, *only the low-order eight bits* are written.
Here is a short example that uses `write( )` to output the character '`A`' followed by a newline to the screen:

```
// Demonstrate System.out.write().
class WriteDemo {
     public static void main(String args[]) {
          int b;
          b = 'A';
          System.out.write(b);
          System.out.write('\n');
          System.out.write(97);
          System.out.write('\n');
          System.out.write(1889); //what is in lower 8-bits?
          System.out.write('\n');
     }
}
```

Note: `print()` & `println()` are handy to use.

PrintWriter Class

The use of System.out is probably best for debugging purposes
For real- world programs, the recommended method of writing to the console is through a `PrintWriter` *stream*. `PrintWriter` is one of the character-based classes.

Using a character-based class for console output makes internationalising your program easier.

`PrintWriter` defines several constructors, including `PrintWriter(OutputStream outputStream,`
`boolean flushingOn)`

Here, `OutputStream` is an object of type `OutputStream`, and `OutputStream` controls whether Java flushes the output stream every time a `println( )` method (among others) is called:

`true` - flushing automatically takes place
`false` – **not** automatic

`PrintWriter` supports the `print( )` and `println( )` methods.
They can be used as used with `System.out`.
If an argument is not a simple type, the `PrintWriter` methods call the object's `toString( )` method and then display the result. To write to the console by using a `PrintWriter`, specify `System.out` for the output stream and automatic flushing.

For example, this line of code creates a `PrintWriter` that is connected to console output:

```
PrintWriter pw = new PrintWriter(System.out, true);
```

The following application illustrates using a `PrintWriter` to handle console output:

```
// Demonstrate PrintWriter
import java.io.*;
public class PrintWriterDemo {
     public static void main(String args[]) {
          PrintWriter pw = new PrintWriter(System.out, true);
          String st1 = "This is a string";
          pw.println(st1);
          int i = -7;
          pw.println(i);
          double d = 4.5e-7;
          pw.println(d);
          pw.printf("A string:%s\nAn integer:%14d \nA double:
%16.8f\n",st1, i, d);
     }
}
```

Using a `PrintWriter` makes your real-world applications easier to internationalise. Otherwise, `System.out` can be used directly as we usually do in sample programs.

Reading and Writing Files

Two of the most often-used stream classes are `FileInputStream` and `FileOutputStream`, which create *byte streams* linked to files.

To open a file, simply create an object of one of these classes, specifying the name of the file as an argument to the constructor.

For example:

```
FileInputStream(String fileName) throws FileNotFoundException
FileOutputStream(String fileName) throws FileNotFoundException
```

Here, `fileName` specifies the name of the file that you want to open.

When you create an input stream, if the file does not exist, then `FileNotFoundException` is thrown.

For output streams, if the file cannot be opened or created, then `FileNotFoundException` is thrown.

`FileNotFoundException` is a subclass of `IOException`.
When an output file is opened, any *preexisting file* by the same name is destroyed.

Once processing a file is done, it must be closed,
by calling the `close( )` method,
which is implemented by both `FileInputStream` and `FileOutputStream,` as

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

To read from a file, you can use a version of `read( )` that is defined within `FileInputStream.`

Example:

```
int read( ) throws IOException
```

It reads single byte from the file and return the byte as an integer value.
`read( )` returns –1 when the end of the file is encountered. It can throw an `IOException`.

Let us display contents of a file:

```
/* Display a text file.
   To use this program, specify the name
   of the file that you want to see.
   For example, to see a file called inputFile1.txt,
   use the following command line.
   java ShowFile inputFile1.txt
 */
import java.io.*;
class ShowFile {
     public static void main(String args[])
     {
          int i;
          FileInputStream fin;
          // First, confirm that a filename has been specified.
          if(args.length != 1) {
               System.out.println("Usage: ShowFile filename");
               return; }
          // Attempt to open the file.
          try {
               fin = new FileInputStream(args[0]);
          } catch(FileNotFoundException e) {
               System.out.println("Cannot Open File");
               return; }
          // At this point, the file is open and can be read.
          // The following reads characters until EOF is
encountered.
          try {
               do {
                    i = fin.read();
                    if(i != -1) System.out.print((char) i);
               } while(i != -1);
          } catch(IOException e) {
               System.out.println("Error Reading File");
          }
          // Close the file.
          try {
               fin.close();
          } catch(IOException e) {
               System.out.println("Error Closing File");
          }
     }
}
```

Writing to a File

To write to a file, the write( ) method defined by FileOutputStream can be used.

Its simplest form is shown here:

```
void write(int byteval) throws IOException
```

This method writes the byte specified by `byteval` to the file. Although `byteval` is declared as an integer, only *the low-order eight bits* are written to the file. If an error occurs during writing, an `IOException` is thrown.

The next example uses write( ) to copy a file:

```java
/* Copy a file.
   To use this program, specify the name
   of the source file and the destination file.
   For example, to copy a file called FIRST.TXT
   to a file called SECOND.TXT, use the following
   command line.
   java CopyFile FIRST.TXT SECOND.TXT
 */
import java.io.*;
class CopyFile {
     public static void main(String args[]) throws IOException
     {
          int i;
          FileInputStream fin = null;
          FileOutputStream fout = null;
          // First, confirm that both files have been specified.
          if(args.length != 2) {
               System.out.println("Usage: CopyFile from to");
               return;
          }
          // Copy a File.
          try {

               // Attempt to open the files.
               fin = new FileInputStream(args[0]);
               fout = new FileOutputStream(args[1]);
               do {
                    i = fin.read();
                    if(i != -1) fout.write(i);
               } while(i != -1);
          } catch(IOException e) {
               System.out.println("I/O Error: " + e);
          } finally {
               try {
                    if(fin != null) fin.close();
               } catch(IOException e2) {
                    System.out.println("Error Closing Input 
File");
               }
               try {
                    if(fout != null) fout.close();
               } catch(IOException e2) {
                    System.out.println("Error Closing Output 
File");
               }
          }
     }
}
```

--x--