Java Programming

Lecture-12

Input/Output

File Handling

I/O Basics

Streams

- Java programs perform I/O through streams.
- A ***stream*** is an abstraction that either 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, even if the actual physical devices to which they are linked differ.

- Thus, the same I/O classes and methods can be applied to different types of devices.

- This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket.

- Likewise, an output stream may refer to the console, a disk file, or a network connection.

- Streams are a clean way to deal with input/ output without having every part of your code understand the difference between a keyboard and a network, for example.

- Java implements streams within class hierarchies defined in the **java.io** package.

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.

The Byte Stream Classes

- Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream** and **OutputStream**.

- Each of these abstract classes has several concrete subclasses that handle the differences among various devices, such as disk files, network connections, and even memory buffers.

- Subclasses are shown in table on next slide.

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

Table 13-1 The Byte Stream Classes in **java.io**

The Character Stream Classes

- Character streams are defined by using two class hierarchies. At the top are two abstract classes: **Reader** and **Writer**.

- These abstract classes handle Unicode character streams.

- Java has several concrete subclasses of each of these. They are shown in table on next slide.

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	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 stream that contains **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

Table 13–2 The Character Stream I/O Classes in **java.io**

The Predefined Streams

- Java programs automatically import the **java.lang** package.
- This package defines a class called **System**, which encapsulates several aspects of the run-time environment.
- **System** also contains three predefined stream variables: **in**, **out**, and **err**.
- These fields are declared as **public**, **static**, and **final** within **System**.
- This means that they can be used by any other part of your program and without reference to a specific **System** object.

- **System.out** refers to the standard output stream. By default, this is the console.
- **System.in** refers to standard input, which is the keyboard by default.
- **System.err** refers to the standard error stream, which also is the console by default.
- However, these streams may be redirected to any compatible I/O device.
- **System.in** is an object of type **InputStream**; **System.out** and **System.err** are objects of type **PrintStream**.

Reading Console Input

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

```
// 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');
  }
}
```

Writing Console Output

```
// Demonstrate System.out.write().
class WriteDemo {
  public static void main(String args[]) {
    int b;

    b = 'A';
    System.out.write(b);
    System.out.write('\n');
  }
}
```

```
// Demonstrate PrintWriter
import java.io.*;

public class PrintWriterDemo {
  public static void main(String args[]) {
    PrintWriter pw = new PrintWriter(System.out, true);

    pw.println("This is a string");
    int i = -7;
    pw.println(i);
    double d = 4.5e-7;
    pw.println(d);
  }
}
```

The output from this program is shown here:

```
This is a string
-7
4.5E-7
```

Reading and Writing Files

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

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

- Although both classes support additional constructors, the following are the forms that we will be using:

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.

- When you are done with a file, you must close it.

- This is done by calling the **close( )** method, which is implemented by both **FileInputStream** and **FileOutputStream**.

- It is shown here:

  void close( ) throws IOException

- Closing a file releases the system resources allocated to the file, allowing them to be used by another file.

- Failure to close a file can result in “memory leaks” because of unused resources remaining allocated.

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

- The one that we will use is shown here:

  int read( ) throws IOException

- Each time that it is called, it reads a single byte from the file and returns the byte as an integer value.

- **read( )** returns –1 when the end of the file is encountered.

- It can throw an **IOException**.

- To write to a file, you can use the **write( )** method defined by **FileOutputStream**.

- Its simplest form is:

  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.

Thank you