Chapter 10:
INPUT AND OUTPUT IN JAVA

Introduction

- The JDK comes with many packages. One such package is the `java.io` package. This package contains many classes in Java which correspond to various ways of performing input and output.

- Although there are many I/O classes, most of them descend from 4 root classes:

```
InputStream          Reader
OutputStream         Writer
```

- `Readers` and `Writers` deal with characters. The `stream` classes deal with bytes. The bytes should be thought of as binary data.

- Each of the above classes are abstract super classes that define the behavior for their subclasses. Each subclass implements this behavior for a specific type of I/O.

- Consult the API Docs for the entire set of methods for each of the classes presented here.

- Before we study the classes which allow actual file I/O, we deal with the `File` class, a class which contains a `String` and information about the file named in the `String`.

  ► This class is meant for querying information about files rather than for performing reads and writes.

The `File` Class

- The program below shows some of the methods from the `File` class.

`FileStatus.java`

```
 1. import java.io.*;
 2. public class FileStatus
 3. {
 4.    public static void main(String args[])
 5.    {
 6.       long len;
 7.       File name;
 8.       for( int i = 0; i < args.length; i++)
 9.       {
10.          name = new File(args[i]);
11.          if( name.exists() )
12.          {
13.             System.out.println(name + " Exists");
14.             len = name.length();
15.             System.out.println("Size: " + len);
16.             if ( name.isDirectory() )
17.                System.out.println("is a dir");
18.          }
19.          else {
20.             System.out.print(name);
21.             System.out.println("Not a file");
22.          }
23.       }
24.    }
25. }
```

- Another useful method from the `File` class is `lastModified()` which returns the time that the file was last modified as the number of milliseconds since January 1, 1970.

Listing Files in a Directory

- The preceding program shows how to determine if a file is a directory. If it is, you may want to perform some action on the files in that directory. Here's a program to list those files.

DirList.java

```
 1. import java.io.*;
 2. public class DirList {
 3.    public static void main(String args[])
 4.    {
 5.       File dir;
 6.       String filename;
 7.       String files[];
 8.       for(int i = 0; i < args.length; i++)
 9.       {
10.          dir = new File(args[i]);
11.          if ( dir.isDirectory())
12.          {
13.             files = dir.list();
14.             for(int j = 0;j < files.length; j++)
15.                System.out.println(files[j]);
16.          }
17.       }
18.    }
19. }
```

I/O Classes

- The `System` class defines three streams:

```
public static final InputStream in;
public static final PrintStream out;
public static final PrintStream err;
```

- `InputStream`, an abstract class, and `PrintStream`, a subclass of `OutputStream` are part of the `java.io` package.

- All subclasses of `InputStream` must provide a `read()` method which delivers the next byte of data from the stream or returns the value `-1` at end of file.

- `InputStream` provides other useful methods for certain types of streams such as `available()` which returns the number of bytes available from the stream and `close()` which closes the stream.

- `PrintStream` provides the many `print` and `println` methods that we have been using throughout the course.

Keyboard Input

- Here's a simple program which gets input from the keyboard using the `read()` method.

Keyboard.java

```
 1. public class Keyboard
 2. {
 3.    public static void main(String args[])
 4.    {
 5.       int val;
 6.       try
 7.       {
 8.          while((val =  System.in.read()) != -1)
 9.             System.out.print((char)val);
10.       }
11.       catch(IOException e)
12.       {
13.          System.err.println("Error: " + e);
14.       }
15.    }
16. }
```

- There are a few items to note in the above code.
  - ► The `read` method can throw an `IOException` and since this is a checked exception, we handle it in the code above.
  - ► The `read` method returns `-1` at the end of the file and thus we need to store the returned value in an `int`.
  - ► However, we don't want to print this value as an `int` so it must be cast to a `char`.

File Streams

- There are over fifty classes in the `java.io` package and thus it is unreasonable to study all of them. We will pick a set of classes that represents most of the I/O that you will need to do in Java.

- We will next look at a pair of classes which allow you to read and write files.

```
FileInputStream extends InputStream
FileOutputStream extends OutputStream
```

- A `FileInputStream` constructor can take either a `File` object, or a `String`.

- A `FileOutputStream` constructor likewise takes either a `File`, or a `String`. There's also a constructor that takes a `String` and a `boolean` value which specifies whether you wish to append to this stream or not.

File Streams

FileRead.java

```
 1. import java.io.*;
 2. public class FileRead
 3. {
 4.    public static void main(String a[])
 5.    {
 6.       int aByte;
 7.       FileInputStream fis = null;
 8.       FileOutputStream fos = null;
 9.       try
10.       {
11.          fis = new FileInputStream(a[0]);
12.          fos = new FileOutputStream(a[1]);
13.          while((aByte =  fis.read()) != -1)
14.             fos.write(aByte);
15.       }
16.       catch(FileNotFoundException e)
17.       {
18.          System.err.println
19.           ("NOT FOUND" + e.getMessage());
20.       }
21.       catch(IOException e)
22.       {
23.          System.err.println
24.           ("Error: " + e.getMessage());
25.       }
26.       finally
27.       {
28.          try
29.          {
30.             fis.close();
31.             fos.close();
32.          }
33.          catch(IOException e)
34.          {
35.          }
36.       }
37.    }
38. }
```

Data Streams

- A `DataInputStream` lets an application read primitive Java data types from an underlying input stream in a machine-independent way. An application uses a `DataOutputStream` to write data that can later be read by a data input stream.

- The constructors in these two classes take neither `File` objects nor `String` objects. They take a `stream` object.

- Thus we run into an interesting situation. We need the methods from `DataInputStream` (or `DataOutputStream`) and the constructor from `FileInputStream` (or `FileOutputStream`).

- Methods from the classes `DataInputStream` and `DataOutputStream` read and write binary data. In general you cannot display these files with utilities such as `cat` (UNIX) and `type` (DOS, Windows) or `Notepad` (Windows).

Data Streams

- Here are some of the methods from `DataOutputStream`.

```
void writeBoolean(boolean v);
void writeChar(int v);
void writeInt(int v);
void writeDouble(double v);
void writeFloat(float v);
void writeInt(int v);
void writeLong(long v);
void writeShort(int v);
```

- Here are some of the methods from `DataInputStream`.

```
void readBoolean(boolean v);
void readChar(int v);
void readInt(int v);
void readDouble(double v);
void readFloat(float v);
void readInt(int v);
void readLong(long v);
void readShort(int v);
```

Data Streams

- The program below writes some integers to the file whose name is given on the command line.

WriteInts.java

```
 1. import java.io.*;
 2. public class WriteInts
 3. {
 4.    public static void main(String a[])
 5.    {
 6.       FileOutputStream fos = null;
 7.       DataOutputStream dos = null;
 8.
 9.       try
10.       {
11.          fos = new FileOutputStream(a[0]);
12.          dos = new DataOutputStream(fos);
13.
14.          for (int i = 1; i <= 10; i++)
15.             dos.writeInt(i);
16.       }
17.       catch(IOException e)
18.       {
19.          System.out.println(e.getMessage());
20.       }
21.       finally
22.       {
23.          try
24.          {
25.             fos.close();
26.             dos.close();
27.          }
28.          catch(IOException e)
29.          {
30.          }
31.       }
32.    }
33. }
```

Data Streams

- The program on the following page reads the data produced by the preceding program. This program is written such that it does not know how many items there are to read.

ReadInts.java

```
 1. import java.io.*;
 2. public class ReadInts
 3. {
 4.    public static void main(String a[])
 5.    {
 6.       FileInputStream fis = null;
 7.       DataInputStream dis = null;
 8.       int val = 0, count = 0;
 9.       try
10.       {
11.          fis = new FileInputStream(a[0]);
12.          dis = new DataInputStream(fis);
13.          while(true) {
14.             val = dis.readInt();
15.             System.out.println(val);
16.             count += 4;
17.          }
18.       }
19.       catch(EOFException eof) {
20.          System.out.println
21.            ("Reached EOF on " + a[0]);
22.       }
23.       catch(IOException e) {
24.          System.out.println(e.getMessage());
25.       }
26.       finally {
27.          System.out.println
28.             (count + " bytes read");
29.          try {
30.             fis.close();
31.             dis.close();
32.          }
33.          catch(IOException e) { }
34.       }
35.    }
36. }
```

Print Streams

- We've been using `PrintStream` objects since the course began.
- `out` is a static `PrintStream` variable of the `System` class.
- In addition to the primitive types, any object can be passed to the `print` or `println` methods.
- When an object is passed to these methods, the `toString` method is called to display the object as a `String`. Thus it is sensible for all classes to override the `toString` method from the `Object` class.
- A `PrintStream` can be associated with a file but the constructor must take a stream.

```
PrintStream ps =
  new PrintStream(new FileOutputStream("out"));
```

- The style of the above is sometimes preferred to the code below.

```
FileOutputStream fos =
  new FileOutputStream("out");
PrintStream ps = new PrintStream(fos);
```

Reading and Writing Objects

- `ObjectInputStream` and `ObjectOutputStream` are used to read and write objects from classes which have implemented the `Serializable` interface, a marker interface.

- Marker interfaces are interfaces with no methods that serve to give information to the compiler. In the case of this marker interface, the information is used to format the writing of a `Serializable` object.

Reading and Writing Objects

Persist.java

```
 1. import java.io.*;
 2. public class Persist
 3. {
 4.    public static void main(String a[])
 5.    {
 6.       FileOutputStream fos = null;
 7.       ObjectOutputStream oos = null;
 8.       Mortgage loans[] =  {
 9.       new Mortgage("Me", 50000.0, .07),
10.       new Mortgage("You”, 60000.0, .08),
11.       new Mortgage("Him", 70000.0, .09)};
12.       try
13.       {
14.          fos = new FileOutputStream(a[0]);
15.          oos = new ObjectOutputStream(fos);
16.          for ( int i = 0; i < loans.length; i++)
17.             oos.writeObject(loans[i]);
18.       }
19.       catch(IOException e)
20.       {
21.          System.err.println("Error: " + e);
22.       }
23.       finally
24.       {
25.          try
26.          {
27.             fos.close();
28.             oos.close();
29.          }
30.          catch(IOException e) {}
31.       }
32.    }
33. }
```

Reading and Writing Objects

- For the preceding code to execute properly, the `Mortgage` class must be defined as:

```
public class Mortgage implements Serializable
{
   ...
   ...
}
```

- When objects are written using `writeObject()`, they must be read using `readObject()` as in the code on the following page. The code should know nothing about the amount of data in the input file and thus it must handle end-of-file through an exception.

Reading and Writing Objects

ReadObjects.java

```
 1. import java.io.*;
 2. public class ReadObjects
 3. {
 4.    public static void main(String a[])
 5.    {
 6.       FileInputStream fis = null;
 7.       ObjectInputStream ois = null;
 8.       Mortgage loan;
 9.       try
10.       {
11.          fis = new FileInputStream(a[0]);
12.          ois = new ObjectInputStream(fis);
13.          while(true)
14.          {
15.             loan = (Mortgage) ois.readObject();
16.             System.out.println(loan);
17.          }
18.       }
19.       catch(EOFException e)
20.       {
21.          System.err.println("EOF on " + a[0]);
22.       }
23.       catch(IOException ioe) {}
24.       catch(ClassNotFoundException cnfe) { }
25.       finally
26.       {
27.          try
28.          {
29.             fis.close();
30.             ois.close();
31.          }
32.          catch(IOException e) {}
33.       }
34.    }
35. }
```

Readers and Writers

- In Java, character data is intended to be handled with subclasses of the two abstract classes `Reader` and `Writer`. One such subclass class is the `FileReader`. Another is `FileWriter`.
- `FileReader` and `FileWriter` are preferred for text files (rather than `FileInputStream` and `FileOutputStream`) because they support 16-bit characters.

Readers and Writers

- Here's a program which shows how to use these classes.

CharReader.java

```
 1. import java.io.*;
 2. public class CharReader
 3. {
 4.    public static void main(String a[])
 5.    {
 6.       FileReader fr = null;
 7.       FileWriter fw = null;
 8.       int val = 0, count = 0;
 9.       try
10.       {
11.          fr = new FileReader(a[0]);
12.          fw = new FileWriter(a[1]);
13.          while((val = fr.read()) != -1)
14.          {
15.             fw.write(val);
16.             count++;
17.          }
18.       }
19.       catch(IOException e)
20.       {
21.          System.out.println(e.getMessage());
22.       }
23.       finally
24.       {
25.          System.out.println
26.           ("Wrote " + count+" chars");
27.          try {
28.             fr.close();
29.             fw.close();
30.          }
31.          catch(IOException e) {}
32.       }
33.    }
34. }
```

Line Input

- Sometimes an application needs to read its input a line at a time. There is a `readLine` method in the `BufferedReader` class. You may have guessed that this class takes a `Reader` for its constructor.

- Thus in order to use the `readLine` method, we first must create a `FileReader`. The skeleton code might look like this.

```
1. FileReader fr = new FileReader("input");
2. BufferedReader br = new BufferedReader(fr);
3.
4. String line;
5.
6. //   readLine returns null at end of file!
7. //
8. while((line = br.readLine()) != null)
9.    System.out.println(line);
```

- The code above is fine for reading lines from a disk file. Now suppose you wanted to read lines from the standard input file.

- Recall that `System.in` is a subclass of `InputStream`. Thus in order to use the `readLine` method, we first must convert the stream `System.in` to a `Reader`. This is where the class `InputStreamReader` comes in handy.

Line Input

- Here's an example of reading lines from the keyboard.

KeyboardReadLines.java

```
 1. import java.io.*;
 2. public class KeyboardLineInput
 3. {
 4.    public static void main(String a[])
 5.    {
 6.       InputStreamReader isr = null;
 7.       BufferedReader br = null;
 8.       String line;
 9.       int n = 0;
10.       try
11.       {
12.          isr = new InputStreamReader(System.in);
13.          br = new BufferedReader(isr);
14.          while((line = br.readLine()) != null)
15.          {
16.             System.out.println(n + "\t" + line);
17.             n++;
18.          }
19.       }
20.       catch(IOException e)
21.       {
22.          System.out.println(e.getMessage());
23.       }
24.       finally
25.       {
26.          try
27.          {
28.             isr.close();
29.             br.close();
30.          }
31.          catch(IOException e) {}
32.       }
33.    }
34. }
```

PrintWriter

- Like `PrintStream`, `PrintWriter` has all the necessary methods to write Java primitive types. A `PrintWriter` constructor can take either a `Writer` or an `OutputStream`. The skeleton code for using a `PrintWriter` is:

```
1. FileOutputStream fos =
2.  new FileOutputStream(args[0]);
3. PrintWriter pw = new PrintWriter(fos);
4. for (int i = 0; i < 100; i++)
5.    pw.println(i);
```

String Readers/Writers

- The `StringReader` and `StringWriter` classes transfer information between strings, not files. The `StringReader` constructor needs a string from which to read data.

- `StringWriter` uses an internal buffer of some default size. If you write to a `StringWriter` object, you can use `getBuffer` to retrieve the internal buffer. Note that `getBuffer` returns a `StringBuffer`.

ReadString.java

```
 1. import java.io.*;
 2. public class ReadString
 3. {
 4.    public static void main(String args[])
 5.    {
 6.       StringReader sr;
 7.       StringWriter sw =new StringWriter();
 8.       int val;
 9.       String input = "data for this example";
10.       StringBuffer output;
11.       try
12.       {
13.          sr = new StringReader(input);
14.          while((val = sr.read()) != -1)
15.             sw.write((char) val);
16.       }
17.       catch(Exception e)       {
18.          System.err.println("Error: " + e);
19.       }
20.       output = sw.getBuffer();
21.       output.reverse();
22.       System.out.println(output);
23.    }
24. }
```

Random Access Files

- The `RandomAccess` class allows for reading and writing on the same file. This kind of object has the usual read and write methods but also adds the following methods.

```
void seek(long pos) throws IOException;
long getFilePointer() throws IOException;
long length() throws IOException;
```

- These classes read/write data in binary format.

Random Access Files

Random.java

```
 1. import java.io.*;
 2. public class Random
 3. {
 4.    public static void main(String a[])
 5.    {
 6.       RandomAccessFile io = null;
 7.       try
 8.       {
 9.          io = new RandomAccessFile(a[0],"rw");
10.          for (int i = 100; i < 255; i++)
11.             io.writeInt(i);
12.          System.out.println
13.           ("File Size: " + io.length());
14.          for ( int i = 1; i < a.length; i++)
15.          {
16.             int val = Integer.parseInt(a[i]);
17.             io.seek((val - 1 ) * 4);
18.             System.out.println(io.readInt());
19.          }
20.       }
21.       catch(IOException e)
22.       {
23.          System.out.println("ERROR" + e);
24.       }
25.       finally
26.       {
27.          try {
28.             io.close();
29.          }
30.          catch(IOException e) {}
31.       }
32.    }
33. }
```

Exercises

1. Write a program which reads a file and copies the file to the standard output except it folds long lines.

   ► The program should get the wrapping number from the command line.

   ► If no file name is given, then the program should read the standard input.

```
java FoldLines 10
this is
this is
this is one of the long lines
this is on
e of the l
ong lines
```

2. Write a program which prints the number of characters, words, and lines in a file named on the command line.

   ► If the file is a directory, then the program should print the number of characters, words, and lines for all files in the directory.

3. Start with `starters/10/3`.

   ► Take some `Mortgages`, serialize them, and write them to a file which is given as the first command line argument.

   ► Then read the `Mortgages` and display only those whose amount is greater than the value given as the second argument on the command line. Do all this in the same program.

Exercises

4. Write a Java Program which inputs a line at a time from the user.

   - The line should contain 3 comma-separated items: a first name, a last name, and a number.
   - You should check each line that you read for correctness.
   - Correct lines should be written as a pipe (`|`) separated record to the file named on the command line. Incorrect lines should be sent to `System.err` along with the reason they are incorrect.
   - End the input process when the user types `end`.

5. Now write a program which reads those lines from the file you created in the previous problem and displays those lines whose number is greater than what is given on the command line.

```
java Problem5 number filename
```

This Page Intentionally Left Blank