I/O Streams

I/O Fundamentals

- The basic units of Input/Output in Java are Streams
- A stream can be thought of as a flow of data from a source to a sink
- Source stream initiates the flow of data (also called as InputStream)
- Sink stream terminates the flow of data (also called OutputStream)
- Sources and sinks are node streams

Types of node streams

- Types of node streams are:
  - Files
  - Memory
  - Pipes (primarily used between threads and processes)

Data within streams

- Java supports two types of streams
  - Character streams
  - Byte streams
- Input and output of character data are handled by readers and writers
- Input and output of byte data are handled by input streams and output streams
  - The term stream refers to byte stream
  - The terms readers and writers refer to character data

Data within streams

	Byte Streams	Character Streams
Source Streams	**InputStream**	**Reader**
Sink Streams	**OutputStream**	**Writer**

InputStream Methods

- Three basic read methods
  - int read ()
  - int read (byte[] buffer)
  - int read (byte[] buffer, int offset, int length)
- Other methods
  - void close ()
  - int available () //bytes that can be immediately read
  - skip (long n)
  - boolean markSupported () //mark and reset are operational
  - void mark (int readLimit) //marks the point and allocate buffer
  - void reset () //returns stream to the marked point

OutputStream Methods

- Three basic read methods
  - void write (int c)
  - void write (byte[] buffer)
  - void write (byte[] buffer, int offset, int length)
- Other methods
  - void close ()
  - void flush () //forces writes to happen
- You should close output streams when you are finished with them

DemoFile.java

```
1   import java.io.*;
2   public class DemoFile
3   {
4    public static void main(String[] args) {
5     try
6      {
     // create a inputstream object to read from a file
7         FileInputStream in  = new FileInputStream(args[0]);
8   // create a outputstream object to write into a file
9          FileOutputStream out = new FileOutputStream(args[1]);
10       int v;
11       while ((v = in.read()) != -1)
12      out.write(v);
13
14       in.close();
15         out.close();
16    }
17    catch (IOException io)
18    {
19    System.err.println("**An IO problem occurred " + io);
20    }
21    }
22  }
```

Reader Methods

- Three basic read methods
  - void read (int c)
  - void read (char[] cbuffer)
  - void read (char[] cbuffer, int offset, int length)
- Other methods
  - void close ()
  - boolean ready ()
  - skip (long n)
  - boolean markSupported ()
  - void mark (int readAheadLimit)
  - void reset ()

Writer Methods

- Three basic read methods
  - void write (int c)
  - void write (char[] cbuffer)
  - void write (char[] cbuffer, int offset, int length)
  - void write (String string)
  - void write (String string, int offset, int length)
- Other methods
  - void close ()
  - void flush ()

Node Streams

- Java has three basic types
  - File streams
  - Memory streams
  - Pipe streams
- Possible to create new node stream classes but you have to deal with native function calls

Node Streams

Type	Character Streams	Byte Streams
File	**FileReader** **FileWriter**	**FileInputStream** **FileOutputStream**
Memory: Array	**CharArrayReader** **CharArrayWriter**	**ByteArrayInputStream** **ByteArrayOutputStream**
Memory: String	**StringReader** **StringWriter**	
Pipe	**PipedReader** **PipedWriter**	**PipedInputStream** **PipedOutputStream**

Reader/Writer Example

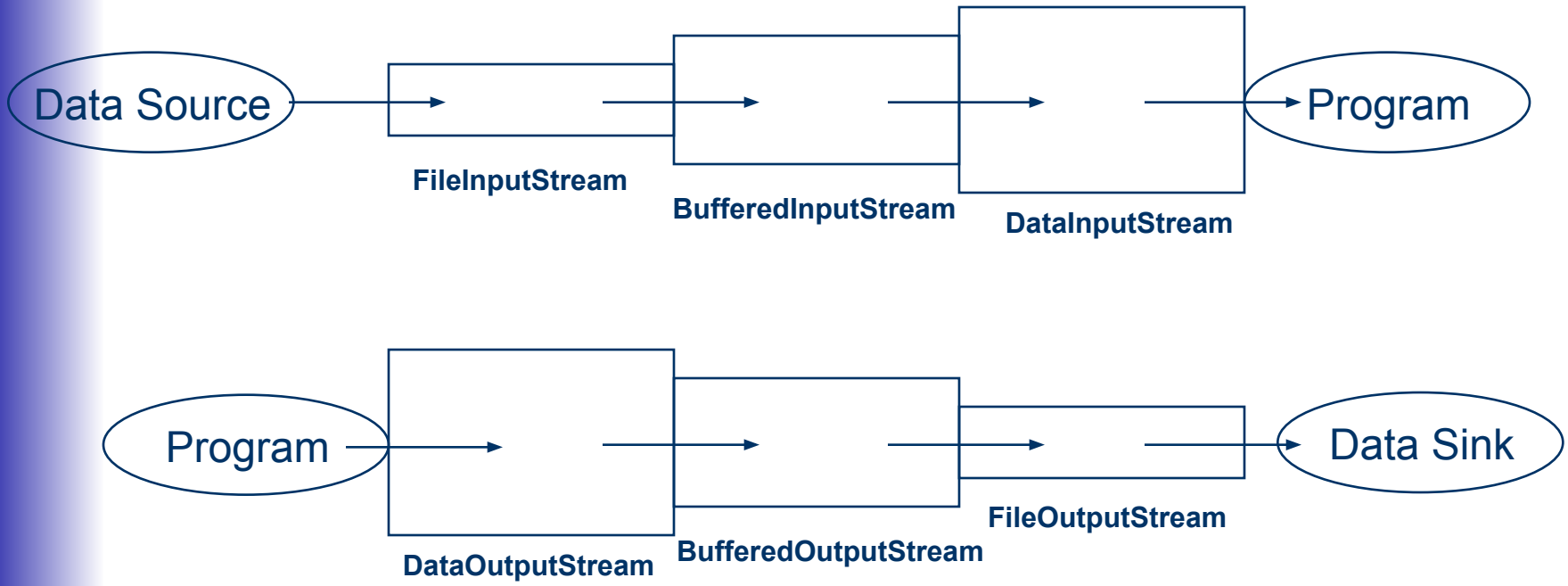
```
import java.io.*;
public class TestNodeStream {
        public static void main (String[] args) {
                try {   FileReader input = new FileReader (args[0]);
                        FileWriter output = new FileWriter (args[1]);
                        char[] buffer = new char [128];
                        int charsRead;
                        charsRead = input.read (buffer);                //Read first buffer
                        while (charsRead != -1) {
                        output.write (buffer, 0, charsRead);   //Write buffer to output file
                        charsRead = input.read (buffer);                //Read the next buffer
                        }
                        input.close ();
                        output.close ();
                }catch (IOException ioe) {
                        ioe.printStackTrace ();   }
        }
}
```

Buffered Reader/Writer Example

```
import  java.io.*;
public class TestBufferedStream {
        public static void main (String[] args) {
                try {
                FileReader input = new FileReader (args[0]);
                BufferedReader bufInput = new BufferedReader (input);
        //Chaining of file reader object
                        FileWriter output = new FileWriter (args[1]);
                        BufferedWriter bufOutput = new BufferedWriter (output);
                        String line;
                        line = bufInput.readLine ();                    //Read the first line
                        while (line != null) {
                                bufOutput.write (line, 0, line.length);
                                bufOutput.newLine ();
                                line = bufInput.readLine ();        //Read the next line
                        }
                        input.close ();         output.close ();
                }catch (IOException ioe) {
                        ioe.printStackTrace ();
                }       }       }
```

I/O Stream Chaining

- Illustrated in the previous example
- A program rarely uses a single stream object
- It chains a series of streams to process data



Processing Streams

- Perform some sort of conversion on another stream
- Also known as filter streams
- A filter input stream is created with a connection to an existing input stream
- When you try to read from filer input stream object, it supplies you with characters that came from the original stream
- Conversions then may be performed to convert raw data into a more usable form

Types of Processing Streams

Type	Character Streams	Byte Streams
Buffering	**BufferedReader** **BufferedWriter**	**BufferedInputStream** **BufferedOutputStream**
Filtering	***FilterReader*** ***FilterWriter***	***FilterInputStream*** ***FilterOutputStream***
Converting between bytes & characters	**InputStreamReader** **OutputStreamWriter**	
Object Serialization		**ObjectInputStream** **ObjectOutputStream**
Data Conversion		**DataInputStream** **DataOutputStream**
Counting	**LineNumberReader**	**LineNumberInputStream**
Peeking ahead	**PushbackReader**	**PushbackInputStream**
Printing	**PrintWriter**	**PrintStream**

Creating a Random Access File

- Often you would want to read data within a file without reading the file from the beginning to end
- Java provides a RandomAccessFile class for handling this type of input/output
- Two options for creating
  - With file name myRAFile = new RandomAccessFile (String name, String mode);
  - With file object myRAFile = new RandomAccessFle (File file, String more);
  - Examples
  - RandomAccessFile myRAFile = new RandomAccessFile (“stock.dbf”, “rw”);

Random Access Files

- RandomAccessFile objects expect to read and write information in the same manner as data input and data output objects
- You have access to all of read () and write () operations found in the DataInputStream and DataOutputStream classes
- long getFilePointer () **//Current location of file pointer**
- void seek (long pos) **//Set file pointer to a specified position**
- long length () **//Length of the file**

Console Input output

- For User interaction with application we required console I/O.
  - Writing to the standard output, the console that launched the application, is achieved using **`System.out`**
    - **`System.out`** is of type **`PrintStream`**,
    - so it allows writing to the standard output a variety if data types using either **`println()`** or **`print()`**:
      - **`void println(char)`**
      - **`void println(String)`**
      - **`void println(float)`**
      - **`void println(int)`**

Reading From the Standard Input

- `System.in`, which is an `InputStream` object, accepts user keyboard information and transfers it to the application.
  - `InputStream` reads binary data, so some bridging must be done to convert it to buffered character data.

- InputStreamReader in = new InputStreamReader(System.in);
- BufferedReader bufin = new BufferedReader(in);

ReadKeyboard.java

```
1   import java.io.*;
2   class DemoKeyboard
3   {
4    public static void main(String args[])   {
5    InputStreamReader in = new
6        InputStreamReader(System.in);
7    BufferedReader bufin  = new
8        BufferedReader(in);
9    while(true)
10   {
11    try {
12      System.out.println(bufin.readLine());
13    }
14    catch(IOException e)   {
15      e.printStackTrace();
16    }
17      }
18   }
19  }
```

Serialization

- Ability to read or write a Java object to a stream
- Addition since JDK 1.1
- Saving an object to persistent storage is called persistence
- Java provides a Serializable interface
- Serializable has no methods and is a marker interface

Serialization – Object Graphs

- When an object is serialized, only data of the object are preserved
- If a data variable is an object, data members of the object are also serialized if that object's class is serializable
- The tree of object's data, including these sub-objects constitutes an object graph
- If a serializable object contains reference to non-serializable element, the entire serialization fails
- If the object graph contains a non-serializable object reference, the object can still be serialized if the non-serializable reference is marked "transient"

Writing Objects to Streams

```
import java.io.*;
import java.util.Date;
public class SerializeDate {
        public SerializeDate () {
                Date d = new Date ();
                try {  FileOutputStream f = new FileOutputStream (“date.ser”);
                        ObjectOutputStream s = new ObjectOutputStream (f);
                        s.writeObject (d);
                        s.close ();
                }catch (IOExceptions ioe) {
                        e.printStackTrace ();
                }
        }
        public static void main (String[] args) {
                SerializeDate s = new SerializeDate ();
        }       }
```

Transient Members

- There are two reasons for declaring a data member as transient:
  - The member is an object that does not implement **`Serializable`**.
  - If We don't wish to write the data member value into the stream.
  - salary will not be a part of serialization process

```
public class Emp implements Serializable {
        transient int salary;
}
```

The `File` Class

- The **`File`** class provides a variety of methods which will give more information about a file

- Create **`File`** objects and you can check following properties of a file
  - whether a File object can be read and written to.
  - whether a **`File`** object exists.
  - Get file (and path) names.
  - Get a list of file names inside a directory.
  - Create a Directory.
  - Rename a file.
  - last modification date.
  - Delete the **`File`** object.

```
File f1 = new File("Demo.txt");
```

File Class methods

- boolean delete()
- isDirectory()
- exists() –
- listFiles()
- mkdir() – Creates a directory according to this File object.
- renameTo() - Renames the file denoted by this object.
- getName()
- getParent()
- getPath()
- length
- list()

Assignment

- Create a class Book having details like bcode, bname , price
- Create object of book class and write into a file
- Read same object from the file and Display it on console