Object Oriented Programming

Chapter Six: Files and Streams in Java

Contents

- Introduction
- Streams
- Types of Streams
  - Byte based Streams
  - Character based streams
  - Predefined streams
- Reading Console Input
- Reading and Writing text files
- Reading and Writing binary files
- Java Serialization
  - Serializing objects
  - De Serializing objects

Introduction

- Whenever our programs have produced output
  - it may be sent to either the Java console, a text area, or some other GUI component.
    - In the sense that they reside in the computer's primary memory and exist only so long as the program is running.
  - Or it may be sent to relatively permanent storage medium
- A **file** is a collection of data stored on a disk or on some other relatively permanent storage medium.
- A file's existence **does not depend** on a running program.
- Here, we will learn how to create files and how to perform input and output operations on their data using the Java classes designed specifically for this purpose.

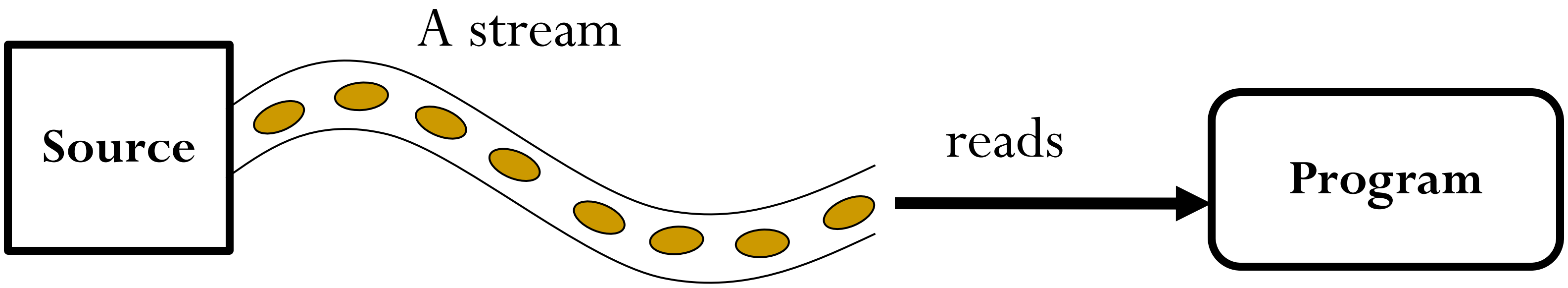
Streams

- **Input** refers to information or data read from some **external source** into a running program.
- **Output** refers to information or data written from a running program to some external destination.
- A disk based file is one possible source, or sink, for data.
- However, data may also be gathered from other sources, e.g. a keyboard, network connection, etc.
- Java employs the notion of a **stream** as an higher level abstraction for all possible data sources.
  - Fundamentally, a stream is a flow of data, never mind where it comes from or goes to.

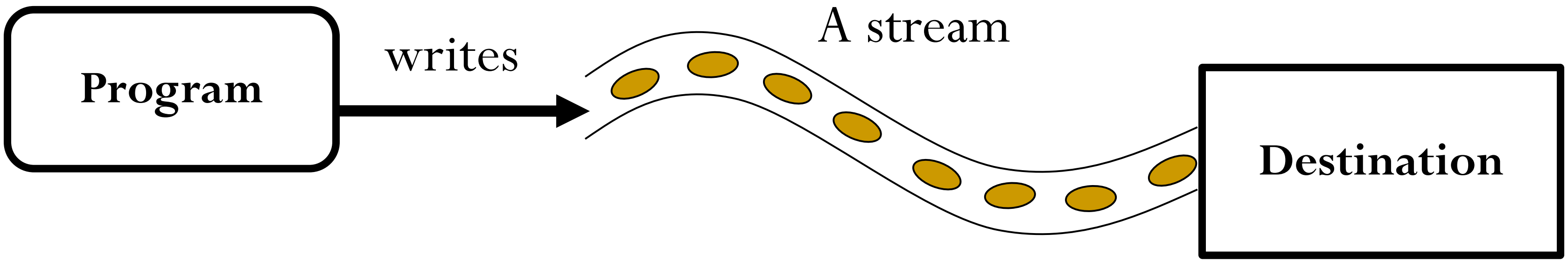
Streams…

- Java input and output is based on the use of streams.

Reading from a stream



Writing to a stream



Types of Streams

- Java defines two types of streams: **byte** based and **character** based.
- **Byte** based Streams provide convenient way for handling input and output of **bytes** and are used for input/output of binary data. Files created by byte based streams are called **binary files**.
  - binary data are not very portable
    - platform dependent
  - On some systems an integer might be 16 bits, and on others it might be 32 bits, so even if you know that a Macintosh binary file contains integers, that still won't make it readable by Windows/Intel programs.
  - But Java binary files are platform independent. They can be interpreted by any computer that supports Java.

Types of Streams…

- **Character** based streams provide a convenient way for handling input and output of characters (**text files**).
- Files created using character based streams are called **text file.**
  - A **text file** is processed as a sequence of characters.
  - A text file created by a program on a Windows/Intel computer can be read by a Macintosh program.
    - Text files are portable
- Note that
  - Text files are human readable files.
  - They are universal and can be edited with many different programs such as NOTEPAD.

Java IO and Streams

- The Java I/O package gives classes support for reading and writing data to and from different input and output sources including Arrays, files, strings, sockets, memory and other data sources.
- The java.io package provides more than 60 input/output classes (stream).
- These classes are used to manipulate binary and text files
- Generally speaking,
  - *binary files* are processed by *subclasses of* InputStream *and* OutputStream.
  - *text files* are processed by *subclasses* of Reader and Writer
- both of which are streams despite their names.

Java's stream hierarchy

Byte based Streams

- Byte based streams classes are defined by two class hierarchies. At the top are two abstract classes:

  1. **InputStream**
  2. **OutputStream**

- Both of these classes are abstract and have several concrete subclasses, that handle the difference between various sources, such as disk, files or sockets.
- The abstract classes **InputStream** and **OutputStream** declare several abstract methods that all subclasses implement.
- Two of the most important are **read( )** and **write( )**, which respectively read and write single byte.

Character based Streams

- Character based streams are also defined by two class hierarchies. At the top are two abstract classes:

  1. **Reader**
  2. **Writer**

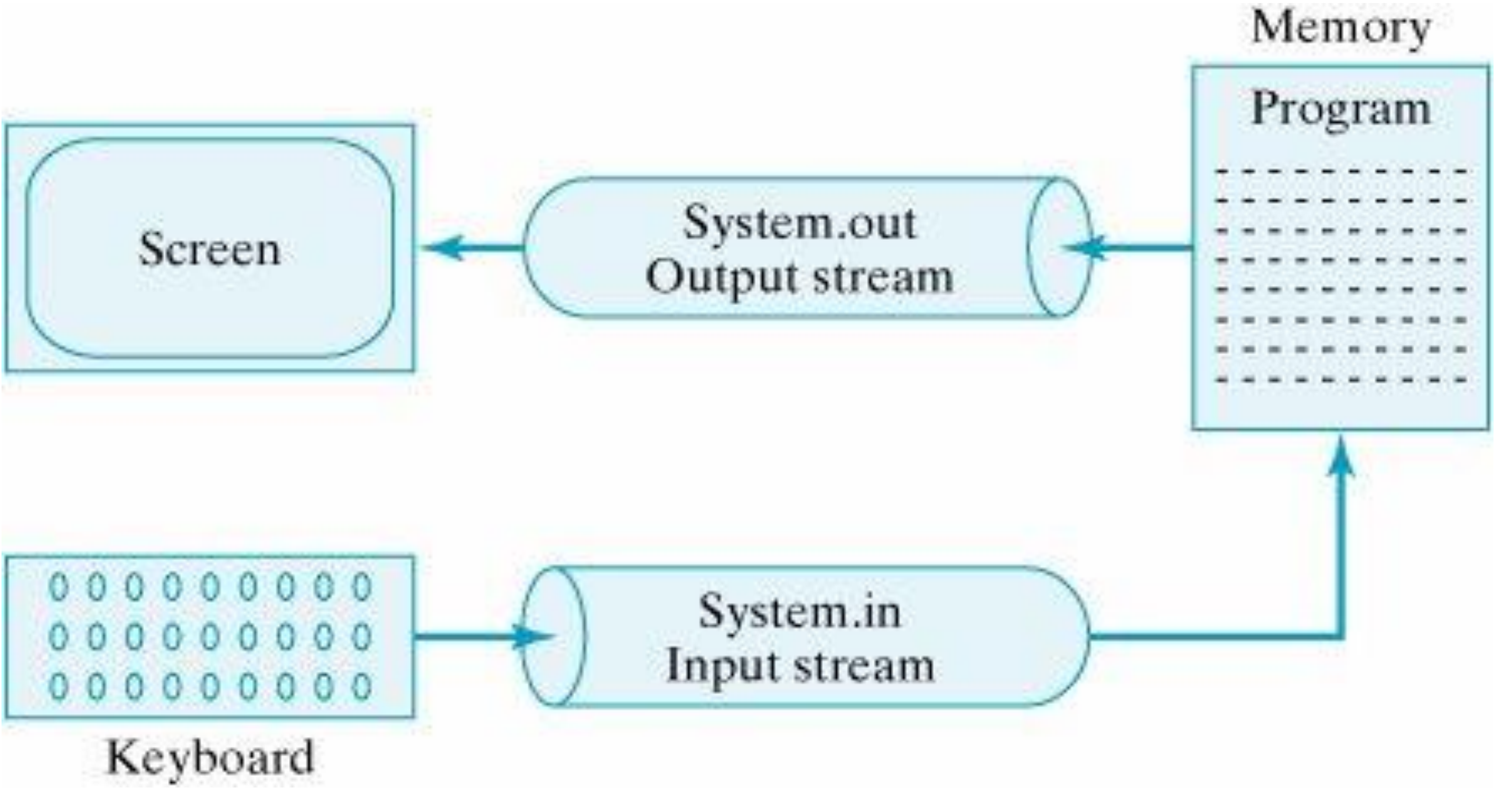
- These abstract classes handle Unicode characters.
- These abstract classes **Reader** and **Writer** declare many abstract methods which are implemented by other subclasses.
- Two most important methods are **read( )** and **write( )**, which respectively read and write one character.
- Both methods are declared as abstract inside **Reader** and **Writer**.

Predefined Streams

- **java.lang** package defines a class called **System**, which encapsulates several aspects of the run time environment: **System.out, System.in, System.err**.
- System class contains three predefined streams, **in**, **out** and **err**.
- They are **public** and **static** fields defined inside the **static final** class **System**.
- **System.out** refers to the standard output stream. By default, this is the console.
- **System.in** refers to the standard input stream, which is the keyboard by default.
- **System.err** refers to the standard error stream, which is also the console by default.

Predefined Streams…

- System.in is an object of **InputStream** class, and **System.out** and **System.err** are objects of **PrintStream** class.

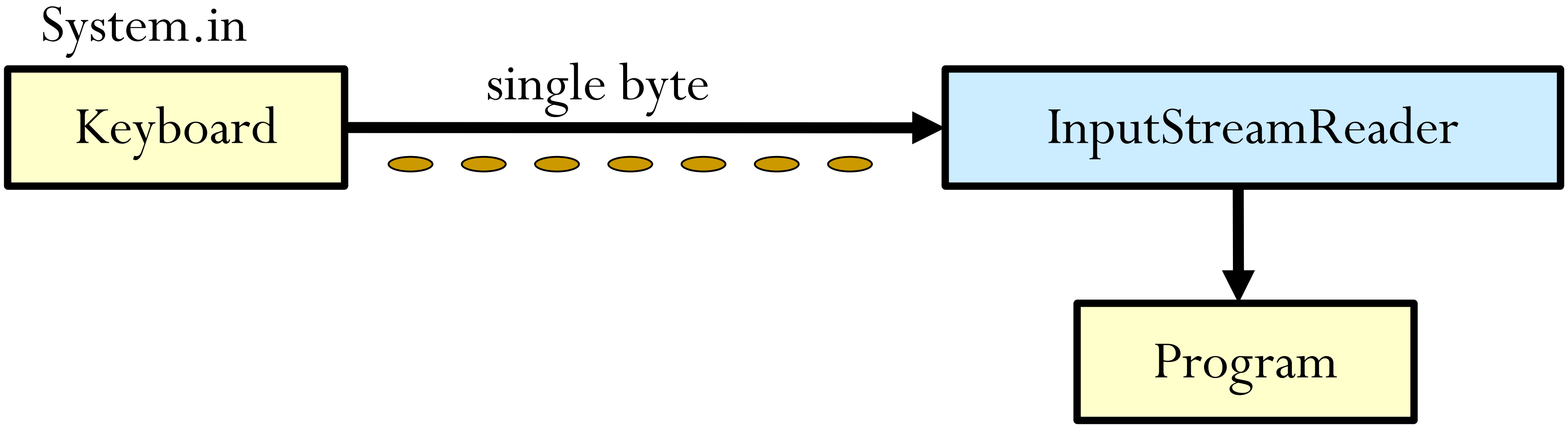


Reading Console Input

- In java, console input is accomplished by reading from **System.in**.
- To read an input we link **System.in** to a **InputStreamReader** class as follows

```
  InputStreamReader r = new
InputStreamReader (System.in);
```

- As **InputStreamReader** read single byte at a time, this affects system performance.



Reading Console Input…

- For this purpose we use **BufferReader** class to read number of bytes from input buffer.
- To read input from buffer we wrap InputStreamReader into BufferedReader as follows:
  - `BufferedReader r  = new BufferedReader(new InputStreamReader ( System.in));`
- BufferedReader has a number of methods.
- By using this methods we can read an input from the keyboard.Some of these methods are:
  - **close()** →Closes the stream and releases any system resources associated with it.
  - **read()** →Reads a single character
  - **readLine()** → Reads a line of text.
  - **ready()** → Tells whether this stream is ready to be read.

The File Class

- Before we see how to read and write binary and text files lets see File class.
- The **File** class does not permit any I/O, instead it provides a means of querying/modifying filename or pathnames (the class would be better termed FileName).
- java.io.File is the central class in working with files and directories.
- Files and directories are both represented by File objects.
- When a File object is created, the system doesn't test to see if a corresponding file/directory actually exists; you must call exists() to check.
- The constructors and methods of the File class are summarized as follows

The File Class…

- Constructors
  - File *f* = new File(String *path*);
    - Create File object for default directory (usually where program is located).
  - File *f* = new File(String *dirpath*, String *fname*);
    - Create File object for directory path given as string.
  - File *f* = new File(File *dir*, String *fname*);
    - Create File object for directory.
- public static constants
  - String *s* = File.separator;
    - Default path separator (eg, "/" in Unix, "\" in Windows).

The File Class…

- *Getting Attributes* (Assume File f)
  - boolean b = f.**exists**(); →true if file exists.
  - boolean b = f.**isFile**(); →true if this is a normal file.
  - boolean b = f.**isDirectory**();→true if f is a directory.
  - String s = f.**getName**(); →name of file or directory.
  - boolean b = f.**canRead**(); →true if can read file.
  - boolean b = f.**canWrite**(); →true if can write file.
  - boolean b = f.**isHidden**(); →true if file is hidden.
  - long l = f.**lastModified**(); →Time of last modification.
  - long l = f.**length**();→Number of bytes in the file.
- *Setting Attributes*
  - *f*.**setLastModified**(*t*); →Sets last modified time to long value *t*.
  - boolean *b* = *f*.**setReadOnly**();→Make file read only. Returns true if successful.

The File Class…

- Paths
  - String s = f.**getPath**();→path name.
  - String s = f.**getAbsolutePath**(); →path name (how is it different from above?).
  - String s = f.**getCanonicalPath**(); →path name. May throw IOException.
  - String s = f.**toURL**();& String s = f.**toURI**(); ; →path with "file:" prefix and /'s. Directory paths end with /.
- Creating and deleting files and directories
  - Boolean b = f.**delete**(); →Deletes the file.
  - boolean b = f.**createNewFile**(); →Create file, may throw IOException. true if OK; false if already exists.
  - boolean b = f.**renameTo**(f2);→Renames f to File f2. Returns true if successful.
  - boolean b = f.**mkdir**();→Creates a directory. Returns true if successful.
  - boolean b = f.**mkdirs**(); →Creates directory and all dirs in path. Returns true if successful.

The File Class… example

```
import java.io.File;
import java.io.IOException;
public class FileTest{
    public static void main(String[] args){
    File f =new File("D:/Documents and
    Settings/esubalew/Desktop/JavaTests/Buffered.java");
    System.out.println(f.exists());
    System.out.println(f.canRead());
    System.out.println(f.canWrite());
    System.out.println(f.getName());
    System.out.println(f.getParent());
    System.out.println(f.isDirectory());
    System.out.println(f.isFile());
    System.out.println(f.length());
    }
}
```

Output:

```
true
true
true
Buffered.java
D:\Documents and
Settings\esubalew\Desktop\JavaTests
false
true
739
```

Writing to a Text File

- Writing data to a file requires three steps:
  1. Connect an output stream to the file.
  2. Write text data into the stream, possibly using a loop.
  3. Close the stream.

- Step 1:
  - The output stream serves as a channel between the program and a named file.
  - The output stream opens the file and gets it ready to accept data from the program.
  - If the file already exists, then opening the file will destroy any data it previously contained.
  - If the file doesn't yet exist, then it will be created from scratch.

Writing to a Text File…

- Step 2
  - Once the file is open, the next step is to write the text to the stream, which passes the text on to the file.
  - This step may require a loop that outputs one line of data on each iteration.
- Step 3
  - Finally, once all the data have been written to the file, the stream should be closed. This also has the effect of closing the file.
  - Even though Java will close any open files and streams when a program terminates normally, it is good programming practice to close the file yourself with a close() statement.
  - This reduces the chances of damaging the file if the program terminates abnormally.

Writing to a Text File…example

- Using FileWriter stream

```
private void writeTextFile(String tobeWritten, String
fileName){
  try{
     FileWriter outStream = new FileWriter (fileName);
      outStream.write (tobeWritten);
      outStream.close();
     }
 catch (IOException e){
    System.out.println("IOERROR:"+e.getMessage()+"\n");
    e.printStackTrace();
     }
} // writeTextFile()
```

Writing to a Text File…

- Constructors of FileWriter stream
  - **FileWriter**(File file)

    Constructs a FileWriter object given a File object.
  - **FileWriter** (File file, boolean append)

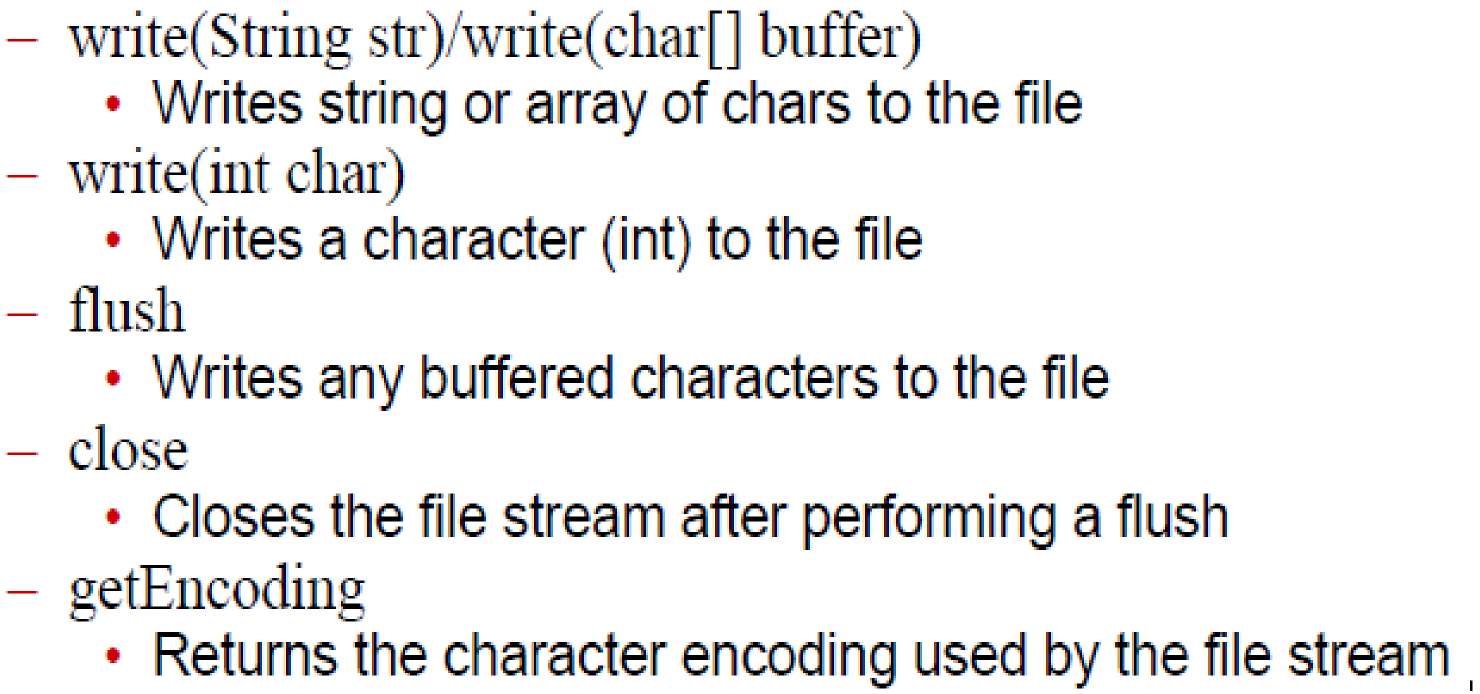
    Constructs a FileWriter object given a File object.
  - **FileWriter**(String fileName)

    Constructs a FileWriter object given a file name.
  - **FileWriter**(String fileName, boolean append)

    Constructs a FileWriter object given a file name with a boolean indicating whether or not to append the data written.

Writing to a Text File…

- Usefull Methods of FileWriter stream



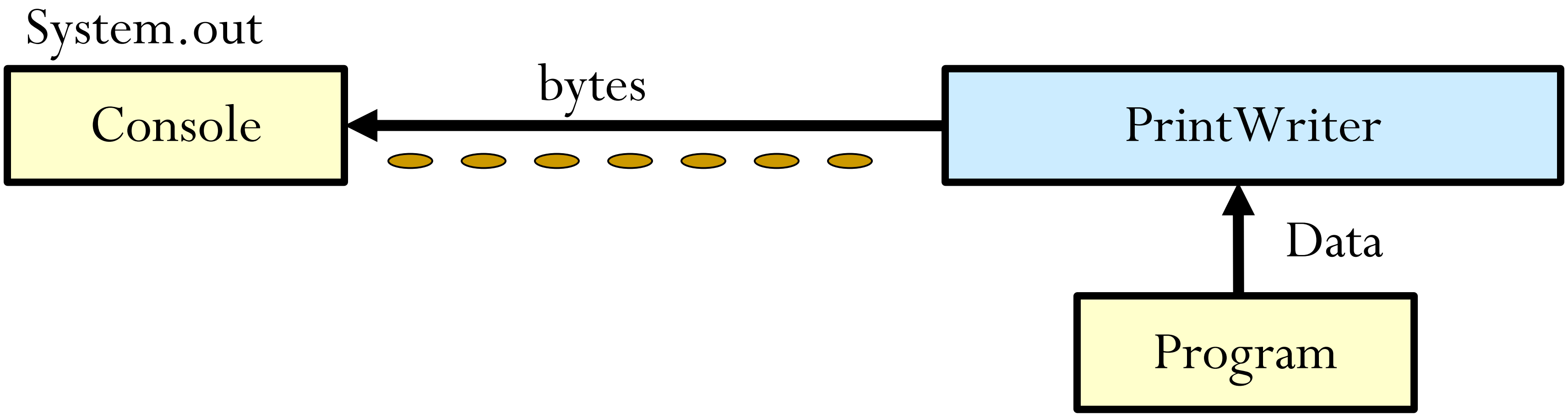
  - write(String str)/write(char[] buffer)
    - Writes string or array of chars to the file
  - write(int char)
    - Writes a character (int) to the file
  - flush
    - Writes any buffered characters to the file
  - close
    - Closes the file stream after performing a flush
  - getEncoding
    - Returns the character encoding used by the file stream

- Other classes which could be used for writing to a text file are: BufferedWritter, CharArrayWritter, PipedWritter, StringWritter, PrintWritter,
- Each of these classes have their own constructors and methods .

PrintWritter

- System.out.println( ) or System.out.write( ) are recommended only for debugging purpose. For real world programs the recommended way of writing to the console us **PrintWriter** class.
- **PrintWriter** is a character based stream. To create the object we use the following constructor.

```
PrintWriter p = new PrintWriter(System.out);
```



PrintWritter…

- **PrintWriter** object takes Strings and data types from program and write in output stream.
- We can use the print( ) and println( ) methods just like System.out.println( ) to write any type of data to the console.

```
    String s = “Hello World”;
    p.println( s );

    int i = 55;
    p.print( i );
```

Reading from a Text File

- Three steps to reading data from a file:
  1. Connect an input stream to the file.
  2. Read the text data using a loop.
  3. Close the stream.
- Some of the classes which are used for reading text files are: all classes are subclasses of Reader class
  - BufferedReader, LineNumberReader
  - CharArrayReader
  - InputStreamReader, FileReader
  - FilterReader
  - PushbackReader
  - PipedReader
  - StringReader

Reading from a Text File…

- Joining a BufferedReader and a FileReader.
- BufferedReader
  - Has readLine() method
  - But lacks a constructor that can take file name
- FileReader
  - Has constructor which can take file name
  - But lacks readLine() method
- Combine them together as follows:

```
BufferedReader inStream = new BufferedReader(new
                                            FileReader(fileName));
```

Reading from a Text File…

- Here the BufferedReader will read from a file
- Now it is possible to use: inStream.readLine() to read one line at a time from the file.
- An important fact about readLine() is that it will return null as its value when it reaches the end of the file.
  - That is readLine() does not return the end-of-line character as part of the text it returns.
- **Exercise**
  - Write a method that takes the file name and extracts each line of the files and displays the line to a consol.

```java
private void readTextFile(String fileName){
 try {
   // Create and open the stream
   BufferedReader inStream = new BufferedReader (new
                                          FileReader(fileName))
                                          ;
   String line = inStream.readLine(); // Read one line
   while (line != null)
     {                        // While more text
      System.out.println(line); // Display a line
      line = inStream.readLine();// Read next line
     }
   inStream.close(); // Close the stream
  }
 catch (FileNotFoundException e){
   System.out.println("IOERROR: "+ fileName +"  NOT  found\n");
   e.printStackTrace();
  }
 catch ( IOException e ){
   System.out.println("IOERROR:"  + e.getMessage() + "\n");
   e.printStackTrace();
  }
} // readTextFile()
```

Copying a file

```
//BufferedReader,  FileReader,  FileWriter,  IOException, PrintWriter;
import java.io.*;
public class FileCopyUsingLines{
  public static void main(String[] args) throws IOException {
    BufferedReader br =null;
    PrintWriter pw=null;
    try{
       br=new BufferedReader(new FileReader(“/source.txt”));
       pw =new PrintWriter(new FileWriter(“/destination.txt”));
       String l;
       while((l=br.readLine())!=null)  pw.println(l);
    }
   finally{
       if(br!=null)  br.close();
       if(pw!=null)  pw.close();
       }
  }
}
```

Reading and Writing Binary Files

- A binary file is a sequence of bytes.
- Unlike a text file, which is terminated by a special end-of-file marker, a binary file consists of nothing but data.
- Just like text files it is possible to read or write binary files using java programs.
- Steps involved in reading and writing binary files are the same as for text files:
  1. Connect a stream to the file.
  2. Read or write the data, possibly using a loop.
  3. Close the stream.

Writing Binary Files

- Let's begin by designing a method that will output employee data to a binary file.
- Let's assume that each record contains three individual pieces of data the employee's name, age, and pay rate.
- For example, the data in a file containing four records might look like this, once the data is interpreted:

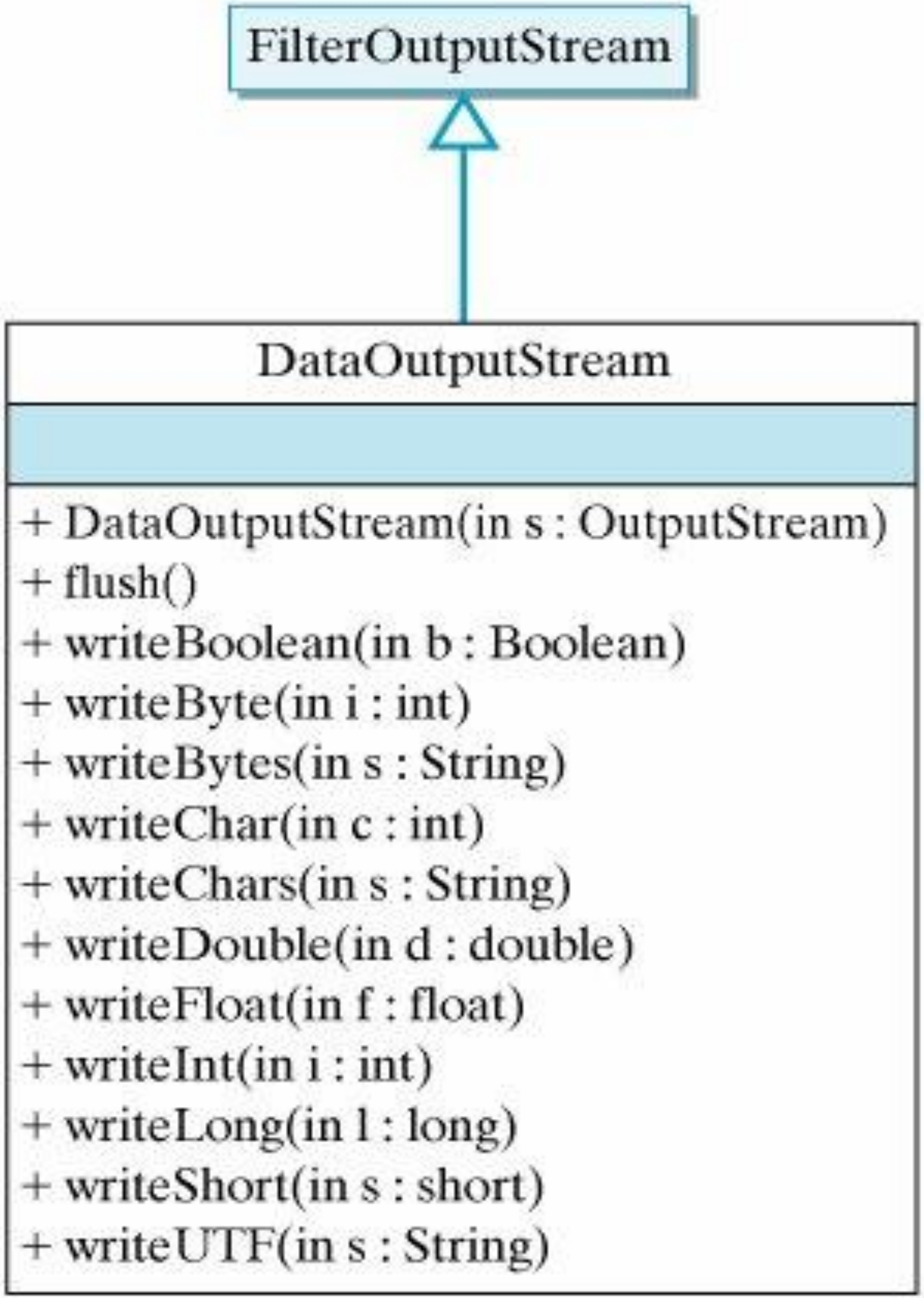
Name0	24	15.06
Name1	25	5.09
Name2	40	11.45
Name3	52	9.25

- Of course, when these data terms are stored in the file or in the program's memory, they just look like one long string of 0's and 1's.

Writing Binary Files…

- We find the answers to these by searching through the java.io package.
- Because we are performing binary output, we need to use a subclass of OutputStream.
- Because we're outputting to a file, one likely candidate is FileOutputStream.
  - This class has the right kind of constructors, but it only contains write() methods for writing ints and bytes.
  - What if we want to write Strings and doubles for instance?
    - These kinds of methods are found in DataOutputStream, which contains a write() method for each different type of data.

Writing Binary Files…



FilterOutputStream

DataOutputStream

```
+ DataOutputStream(in s : OutputStream)
+ flush()
+ writeBoolean(in b : Boolean)
+ writeByte(in i : int)
+ writeBytes(in s : String)
+ writeChar(in c : int)
+ writeChars(in s : String)
+ writeDouble(in d : double)
+ writeFloat(in f : float)
+ writeInt(in i : int)
+ writeLong(in l : long)
+ writeShort(in s : short)
+ writeUTF(in s : String)
```

The **java.io.DataOutputStream** class contains methods for writing all types of data.

Writing Binary Files…

- To construct a stream to use in writing employee records, we want to join together a DataOutputStream and a FileOutputStream.
- The DataOutputStream gives us the output methods we need, and the FileOutputStream lets us use the file's name to create the stream:
- See the following:

```
DataOutputStream outStream = new  DataOutputStream( new
FileOutputStream (fileName));
```

- DataOutputStream has one constructor
  - **DataOutputStream** (OutputStream out): Creates a new data output stream to write data to the specified underlying output stream.

Writing Binary Files… Example

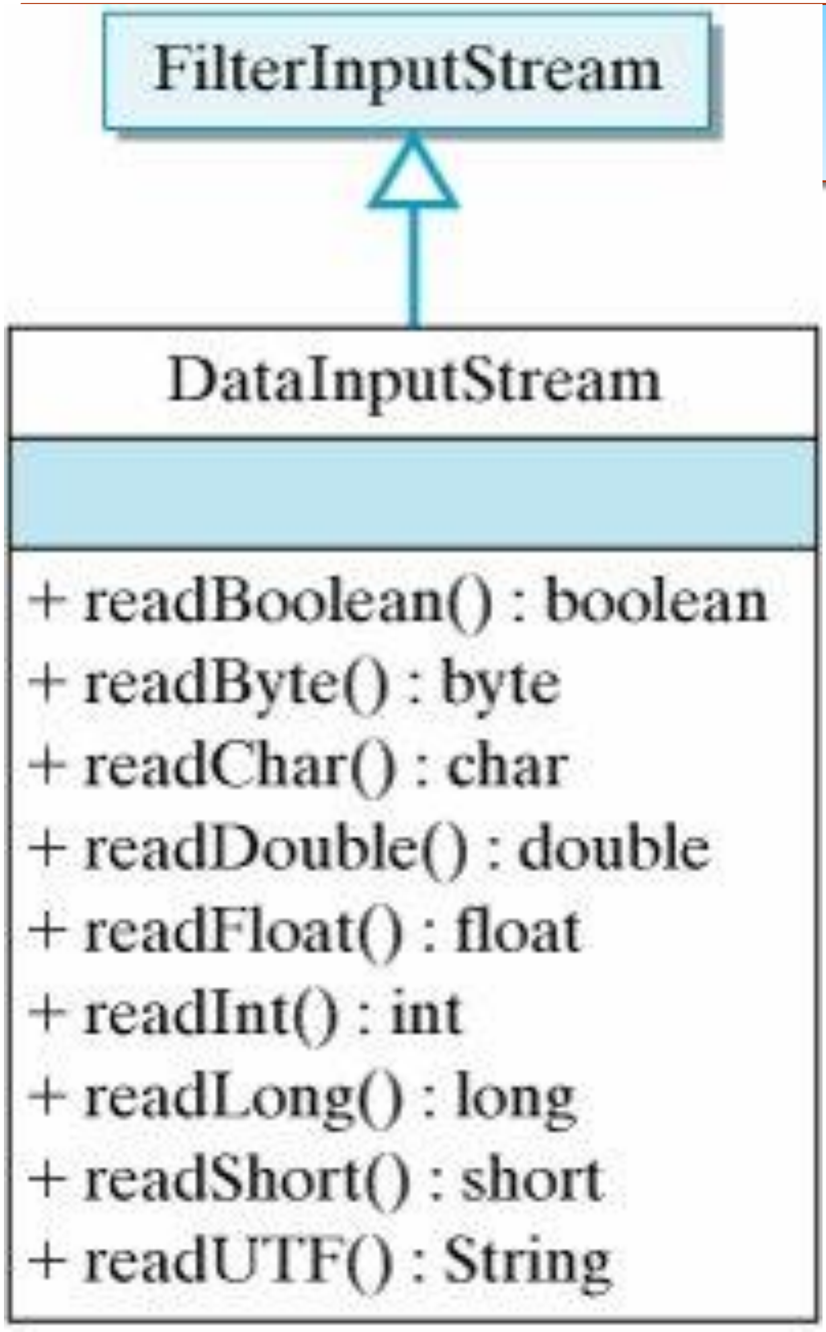
```
import java.io.*;
import java.util.Scanner;
public class Example {
public static void main(String[] args)throws IOException{
          Scanner in =  new Scanner (System.in);
          DataOutputStream outStream = new  DataOutputStream(new
FileOutputStream ("D:/Documents and Settings/esubalew/Desktop/JavaTests/Sample.bin"));
                    for (int i = 0;i<5;i++){
                              System.out.println("Enter Name");
                              outStream.writeUTF(in.next());
                              System.out.println("Enter ID");
                              outStream.writeInt(in.nextInt());
                              System.out.println("Enter GPA");
                              outStream.writeDouble(in.nextDouble());}}
}
```

Reading Binary Files

- The steps involved in reading data from a binary file are the same as for reading data from a text file: create an input stream and open the file, read the data, close the file.
- The main difference lies in the way you check for the end-of-file marker in a binary file.
- For this purpose (reading) combination of DataInputStream and FileInputStream is used.

```
DataInputStream inStream = new DataInputStream(new
                    FileInputStream(file));
```

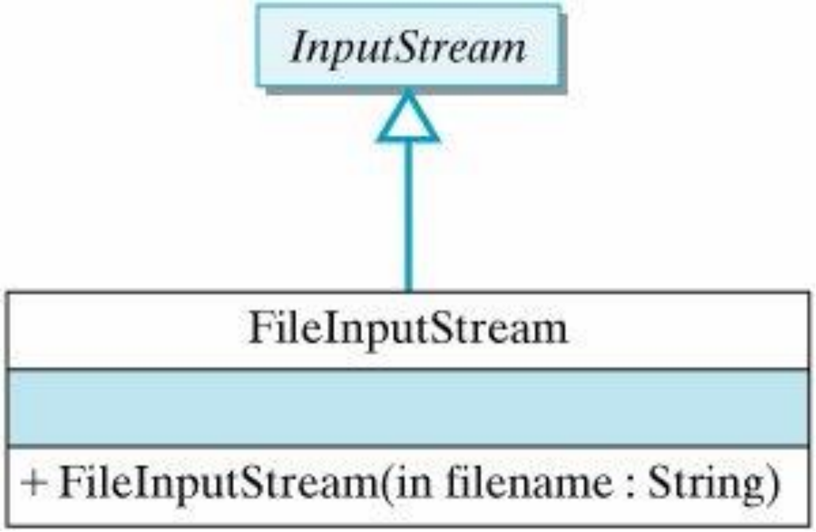
Reading Binary Files…



FilterInputStream

DataInputStream

+ readBoolean() : boolean
+ readByte() : byte
+ readChar() : char
+ readDouble() : double
+ readFloat() : float
+ readInt() : int
+ readLong() : long
+ readShort() : short
+ readUTF() : String



InputStream

FileInputStream

+ FileInputStream(in filename : String)

Reading Binary Files, example

```
import java.io.*;
Public class TestClass{
Public static void main(String[] argv)throws IOException{
   Try{
      DataInputStream inStream = new DataInputStream(new
        FileInputStream(“C:\Sample.bin”)); //open stream
      System.out.println(“Name        ID         GPA\n”);
      Try{
        while(true){
           String name  = inStream.readUTF();
           int id = instream.readInt();
           double gpa = inStream.readDouble();
           System.out.println(name + “  “+id+” “+gpa +” \n”);
           }//while
        }
```

Reading Binary Files, example…

```
    Catch(IOException e){ }
    inStream.close();
    Catch(FileNotFoundException e){
        System.out.println(“IOError : File Not Found\n”);
    }
    Catch(IOException e){
        System.out.println(“IOError :” + e.getMessage() +”\n”);
     }
  }
}
```

Java - Serialization

- Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.
- After a serialized object has been written into a file, it can be read from the file and deserialized
  - that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.
- Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Java – Serialization…

- Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.
- The ObjectOutputStream class contains many write methods for writing various data types, but one method in particular stands out:

```
public final void writeObject(Object x) throws IOException
```

- The above method serializes an Object and sends it to the output stream.

Java – Serialization…

- Similarly, the ObjectInputStream class contains the following method for deserializing an object:

```
public final Object readObject() throws IOException,
                                        ClassNotFoundException
```

- This method retrieves the next Object out of the stream and deserializes it.
- The return value is Object, so you will need to cast it to its appropriate data type.
- To demonstrate how serialization works in Java, suppose that we have the following Employee class, which implements the Serializable interface.

Java – Serialization…Example

```
public class Employee implements java.io.Serializable {
   public String name;
   public String address;
   public int transient SSN;
   public int number;
   public void mailCheck() {
      System.out.println("Mailing a check to " + name+ " " + address);
   }
}
```

- Notice that for a class to be serialized successfully, two conditions must be met:
  - The class must implement the java.io.Serializable interface.
  - All of the fields in the class must be serializable. If a field is not serializable, it must be marked transient.

Serialization - Serializing an Object

- The ObjectOutputStream class is used to serialize an Object. The following SerializeDemo program instantiates an Employee object and serializes it to a file.
- When the program is done executing, a file named employee.ser is created. The program does not generate any output, but study the code and try to determine what the program is doing.
- **Note:** When serializing an object to a file, the standard convention in Java is to give the file a **.ser** extension.

```
import java.io.*;
 public class SerializeDemo{
   public static void main(String [] args)  {
      Employee e = new Employee();
      e.name = "Reyan Ali";
      e.address = "Phokka Kuan, Ambehta Peer";
```

Serialization - Serializing an Object

```
e.SSN = 11122333;
e.number = 101;
try{
      FileOutputStream fileOut =new FileOutputStream("employee.ser");
      ObjectOutputStream out =new ObjectOutputStream(fileOut);
      out.writeObject(e);
      out.close();
      fileOut.close();
    }catch(IOException i){
       i.printStackTrace();
    }
  }
}
```

Serialization - DeSerializing an Object

- The following DeserializeDemo program deserializes the Employee object created in the SerializeDemo program.

```
import java.io.*;
public class DeserializeDemo{
  public static void main(String [] args){
    Employee e = null;
    try{
      FileInputStream fileIn = new FileInputStream("employee.ser");
      ObjectInputStream in = new ObjectInputStream(fileIn);
      e = (Employee) in.readObject();
      in.close();
      fileIn.close();
    }catch(IOException i){
      i.printStackTrace();
      return;
    }
```

Serialization - DeSerializing an Object…

```
catch(ClassNotFoundException c){
        System.out.println(.Employee class not found.);
        c.printStackTrace();
        return;
      }
      System.out.println("Deserialized Employee...");
      System.out.println("Name: " + e.name +“ , Address: " + e.address
    + “, SSN: " + e.SSN + “,  Number: " + e.number);
 }
 }
```

Output

```
Deserialized Employee...
Name: Reyan Ali,  Address:Phokka Kuan, Ambehta Peer, SSN: 0, Number:101
```

Serialization - DeSerializing an Object…

- Here are following important points to be noted:
  - The try/catch block tries to catch a ClassNotFoundException, which is declared by the readObject() method. For a JVM to be able to deserialize an object, it must be able to find the bytecode for the class. If the JVM can't find a class during the deserialization of an object, it throws a ClassNotFoundException.
  - Notice that the return value of readObject() is cast to an Employee reference.
  - The value of the SSN field was 11122333 when the object was serialized, but because the field is transient, this value was not sent to the output stream. The SSN field of the deserialized Employee object is 0.

Important Points about Serialization

- If you declare a variable as **transient** it will not be saved during serialization.
- **Serializable** interface is an empty interface.
- If a class is serializable, then all the subclasses of this super class are implicitly serializable even if they don't explicitly implement the **Serializable** interface.
- If you are serializing an array or collection, each of its elements must be serializable.
- **static** variables are not saved as part of serialization.

Thank You !!!

Class End!!!