Streams and Input/Output Files

Resource: Rajkumar Buyya
Grid Computing and Distributed Systems (GRIDS) Laboratory
Dept. of Computer Science and Software Engineering
University of Melbourne, Australia

Introduction

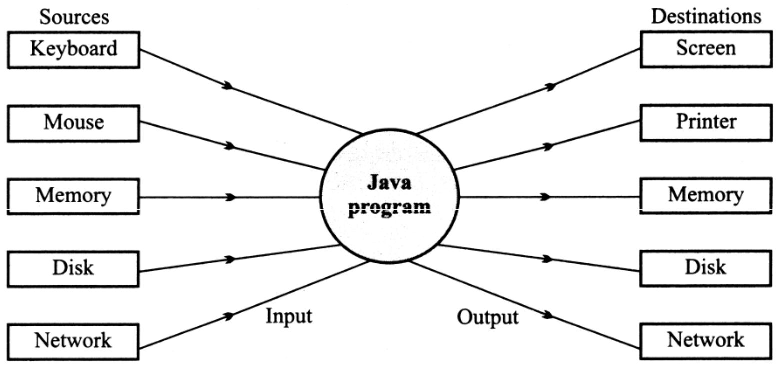
- So far we have used variables and arrays for storing data inside the programs. This approach poses the following limitations:
  - The data is lost when variable goes out of scope or when the program terminates. That is data is stored in temporary/mail memory is released when program terminates.
  - It is difficult to handle large volumes of data.
- We can overcome this problem by storing data on secondary storage devices such as floppy or hard disks.
- The data is stored in these devices using the concept of Files and such data is often called persistent data.

File Processing

- Storing and manipulating data using files is known as file processing.
- Reading/Writing of data in a file can be performed at the level of bytes, characters, or fields depending on application requirements.
- Java also provides capabilities to read and write class objects directly. The process of reading and writing objects is called object serialisation.

I/O and Data Movement

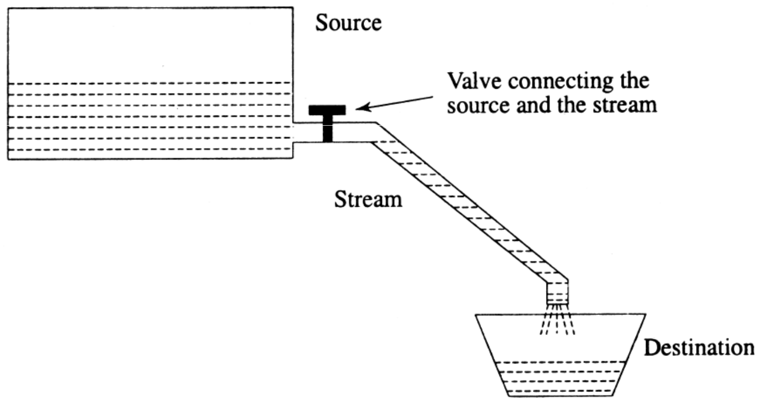
- The flow of data into a program (input) may come from different devices such as keyboard, mouse, memory, disk, network, or another program.
- The flow of data out of a program (output) may go to the screen, printer, memory, disk, network, another program.
- Both input and output share a certain common property such as unidirectional movement of data – a sequence of bytes and characters and support to the sequential access to the data.



Relationship of Java program with I/O devices

Streams

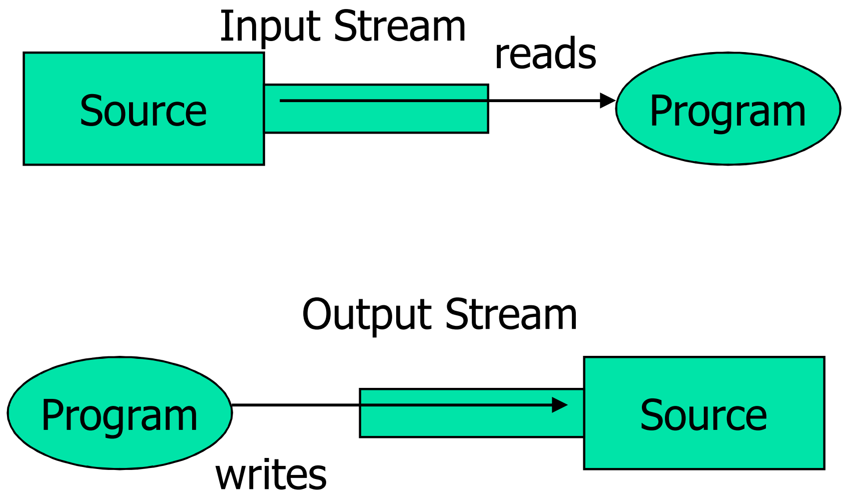
- Java Uses the concept of Streams to represent the ordered sequence of data, a common characteristic shared by all I/O devices.
- Streams presents a uniform, easy to use, object oriented interface between the program and I/O devices.
- A stream in Java is a path along which data flows (like a river or pipe along which water flows).



Conceptual view of a stream

Stream Types

- The concepts of sending data from one stream to another (like a pipe feeding into another pipe) has made streams powerful tool for file processing.
- Connecting streams can also act as filters.
- Streams are classified into two basic types:
  - Input Steam
  - Output Stream



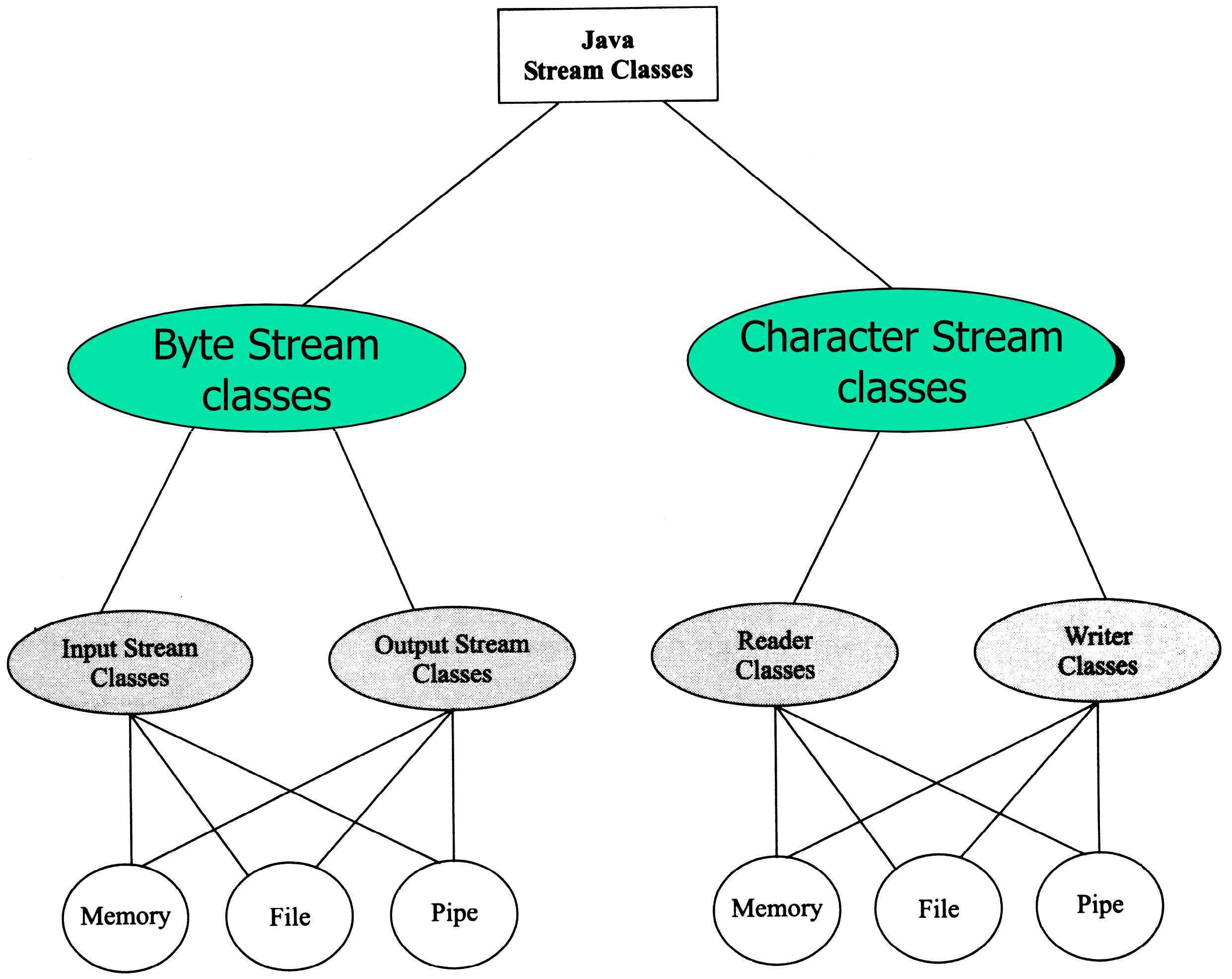
Java Stream Classes

- Input/Output related classes are defined in java.io package.
- Input/Output in Java is defined in terms of streams.
- A *stream* is a sequence of data, of no particular length.
- Java classes can be categorised into two groups based on the data type one which they operate:
  - *Byte streams*
  - *Character Streams*

Streams

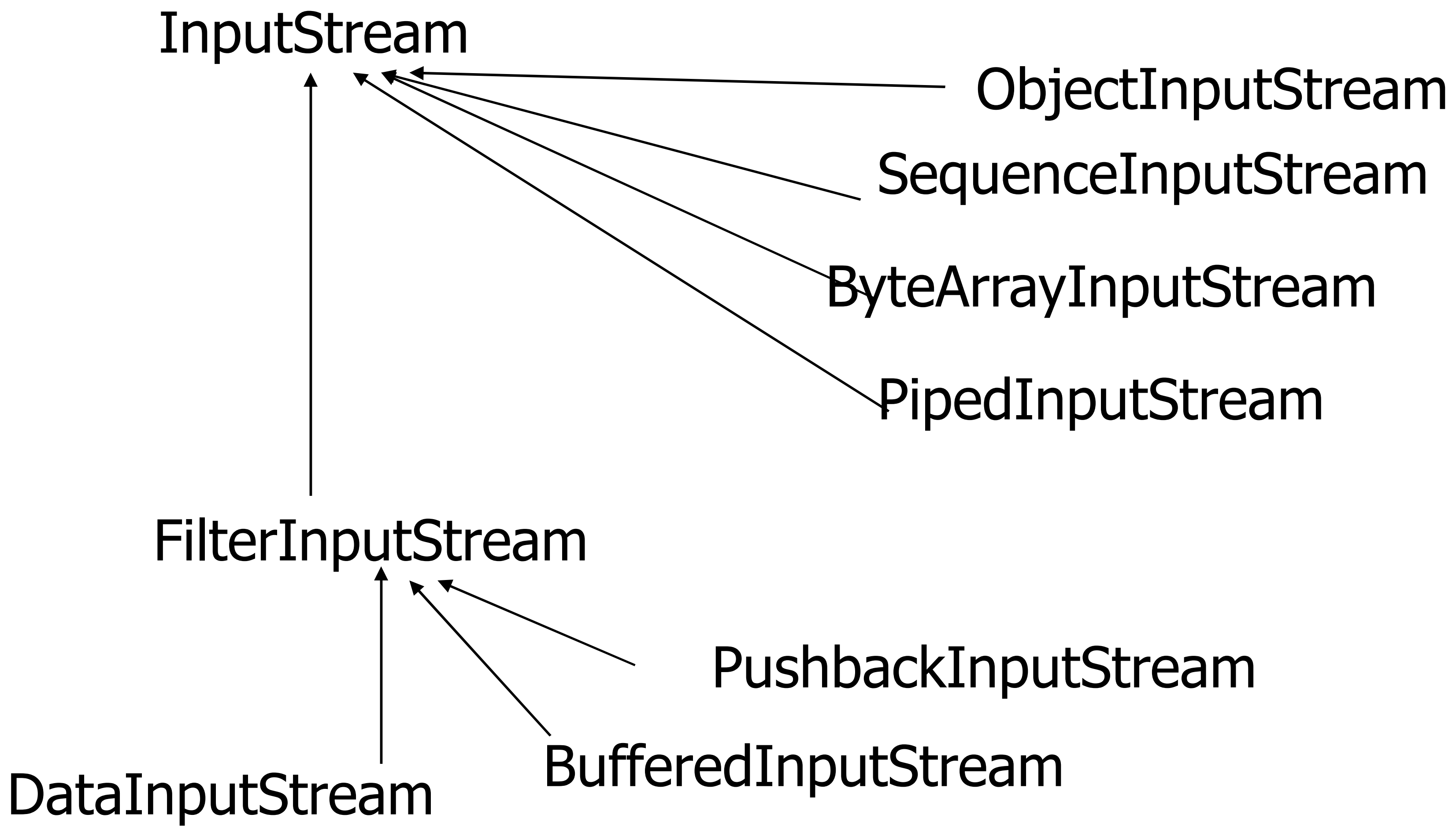
Byte Streams	Character streams
Operated on 8 bit (1 byte) data.	Operates on 16-bit (2 byte) unicode characters.
Input streams/Output streams	Readers/ Writers

Classification of Java Stream Classes



Classification of Java stream classes

Byte Input Streams



Byte Input Streams - operations

public abstract int read()	Reads a byte and returns as a integer 0-255
public int read(byte[] buf, int offset, int count)	Reads and stores the bytes in buf starting at offset. Count is the maximum read.
public int read(byte[] buf)	Same as previous offset=0 and length=buf.length()
public long skip(long count)	Skips count bytes.
public int available()	Returns the number of bytes that can be read.
public void close()	Closes stream

Byte Input Stream - example

- Count total number of bytes in the file

```
import java.io.*;

class CountBytes {
        public static void main(String[] args)
            throws FileNotFoundException, IOException
        {
                FileInputStream in;
                in  =  new FileInputStream(“InFile.txt”);

                int total = 0;
                while (in.read() != -1)
                        total++;
                System.out.println(total + “ bytes”);
        }
}
```

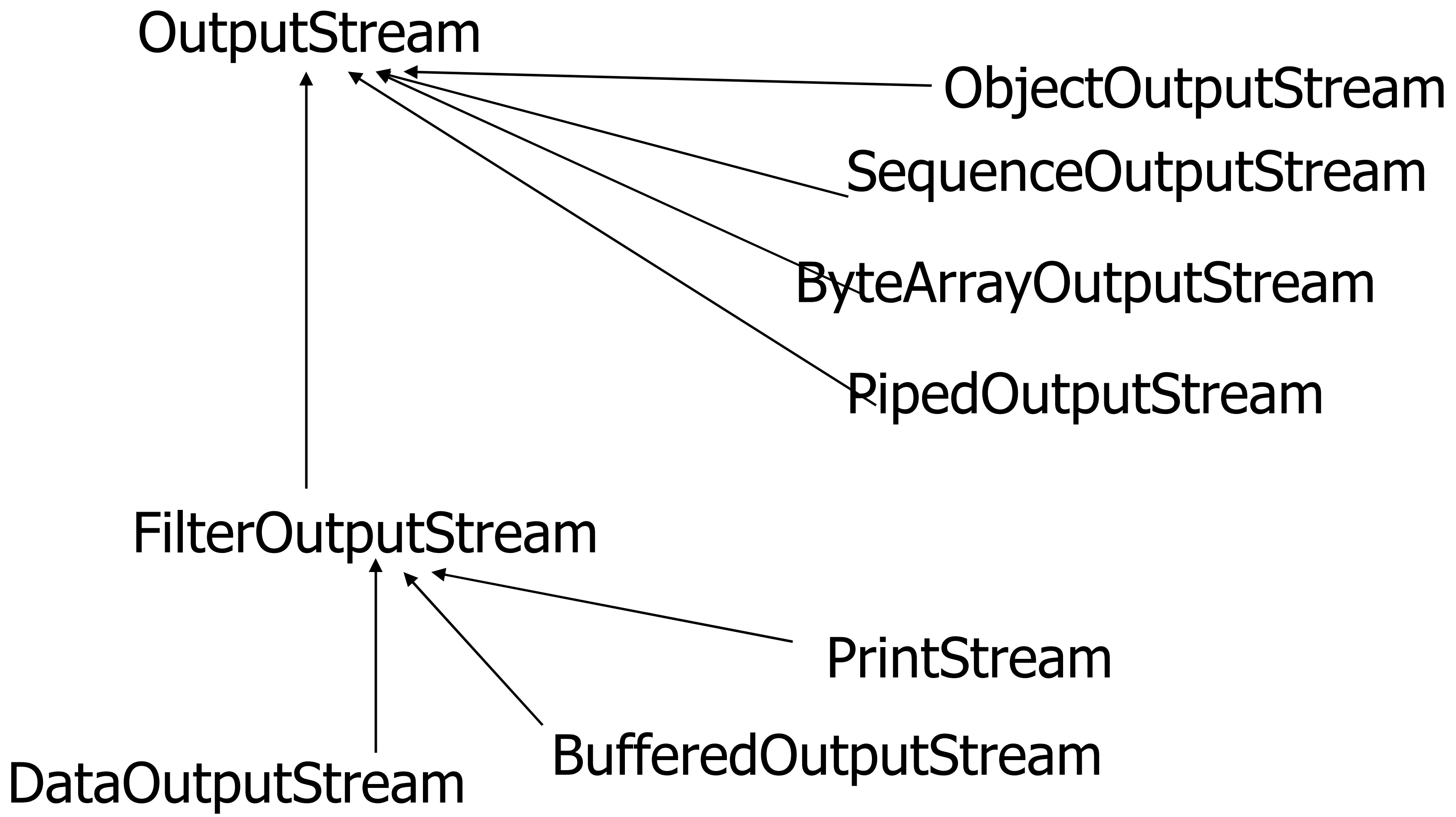
What happens if the file did not exist

- JVM throws exception and terminates the program since there is no exception handler defined.

```
[raj@mundroo] Streams [1:165] java CountBytes

Exception in thread "main" java.io.FileNotFoundException:
   FileIn.txt (No such file or directory)
        at java.io.FileInputStream.open(Native Method)
        at
   java.io.FileInputStream.<init>(FileInputStream.java:64)
        at CountBytes.main(CountBytes.java:12)
```

Byte Output Streams



Byte Output Streams - operations

public abstract void write(int b)	Write *b* as bytes.
public void write(byte[] buf, int offset, int count)	Write *count* bytes starting from *offset* in *buf.*
public void write(byte[] buf)	Same as previous *offset=0* and *count = buf.length()*
public void flush()	Flushes the stream.
public void close()	Closes stream

Byte Output Stream - example

- Read from standard in and write to standard out

```
import java.io.*;

class ReadWrite {
        public static void main(string[] args)
                throws IOException
        {
                int b;
                while (( b = System.in.read()) != -1)
                {
                        System.out.write(b);
                }

}
```

Summary

- Streams provide uniform interface for managing I/O operations in Java irrespective of device types.
- Java supports classes for handling Input Steams and Output steams via java.io package.
- Exceptions supports handling of errors and their propagation during file operations.

Files and Exceptions

- When creating files and performing I/O operations on them, the systems generates errors. The basic I/O related exception classes are given below:
  - EOFException – signals that end of the file is reached unexpectedly during input.
  - FileNotFoundException – file could not be opened
  - InterruptedIOException – I/O operations have been interrupted
  - IOException – signals that I/O exception of some sort has occurred – very general I/O exception.

Syntax

- Each I/O statement or a group of I/O statements much have an exception handler around it/them as follows:

```
try {
…// I/O statements – open file, read, etc.
}
catch(IOException e) // or specific type exception
{
    …//message output statements
}
```

Example

```
import java.io.*;
class CountBytesNew {

	public static void main (String[] args)
		throws FileNotFoundException, IOException  // throws is optional in this case
	{

		FileInputStream in;
		try{
			in = new FileInputStream("FileIn.txt");
			int total = 0;
			while (in.read() != -1)
				total++;
			System.out.println("Total = " + total);
		}
		catch(FileNotFoundException e1)
		{
			System.out.println("FileIn.txt does not exist!");
		}
		catch(IOException e2)
		{
			System.out.println("Error occured while read file FileIn.txt");
		}

	}

}
```

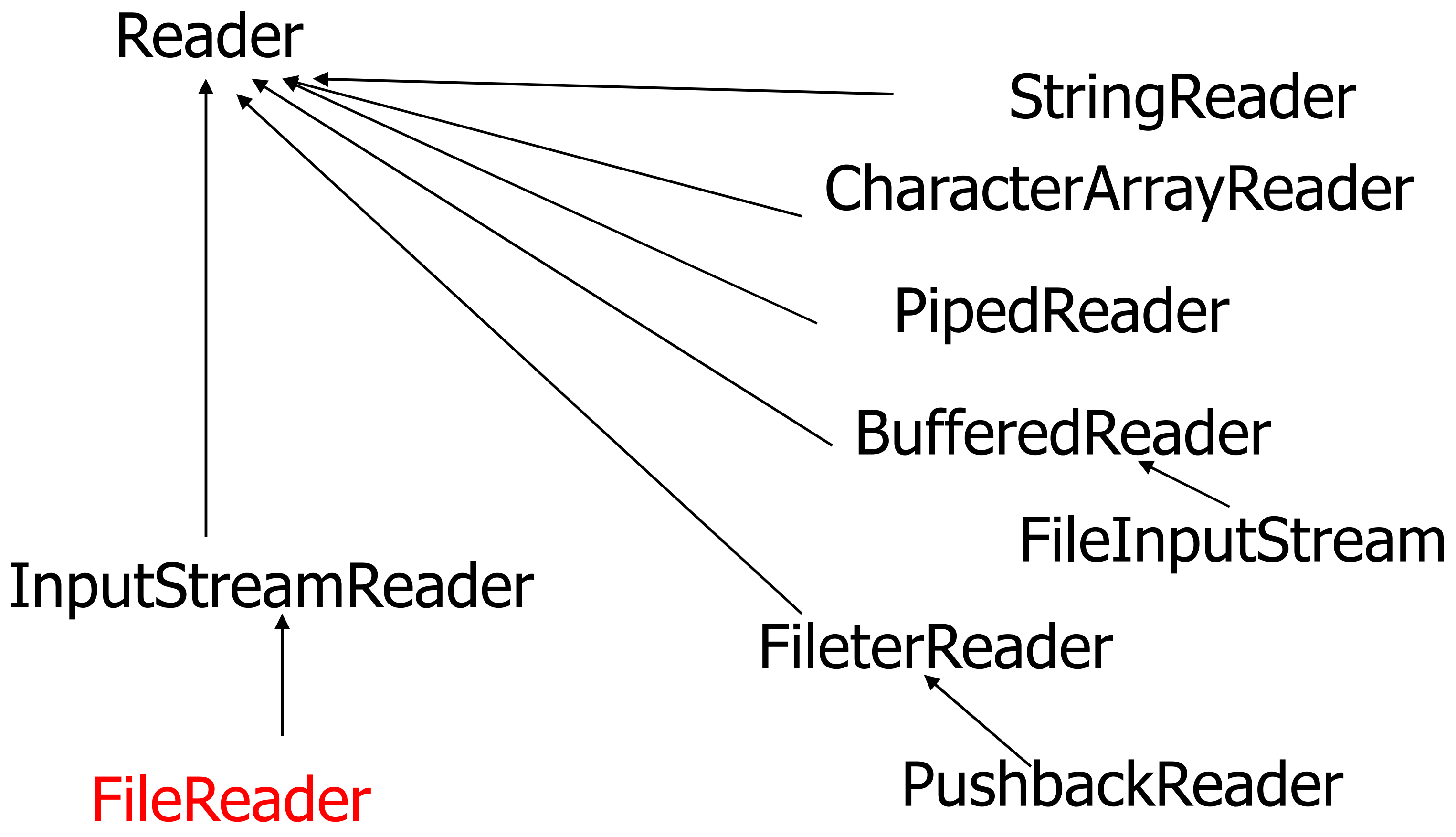
Creation of Files

- There are 2 ways of initialising file stream objects:
  - Passing file name directly to the stream constructor
    - Similar to previous example
  - Passing File Object:
    - Create File Object
      - File inFile = new File(**"FileIn.txt");**
    - **Pass file object while creating stream:**
      - **try {**
        - **in = new FileInputStream(**inFile**);**
      - **}**
- **Manipulation operations are same once the file is opened.**

Reading and Writing Characters

- As pointed out earlier, subclasses of Reader and Writer implement streams that can handle characters.
- The two subclasses used for handling characters in file are:
  - FileReader
  - FileWriter
- While opening a file, we can pass either file name or File object during the creation of objects of the above classes.

Reader Class Hierarchy



Reader - operations

public int read()	Reads a character and returns as a integer 0-255
public int read(char[] buf, int offset, int count)	Reads and stores the characters in *buf* starting at *offset*. *count* is the maximum read.
public int read(char[] buf)	Same as previous *offset*=0 and *length=buf.length*()
public long skip(long count)	Skips *count* characters.
public boolean()	Returns true if the stream is ready to be read.
public void close()	Closes stream

Reader - example

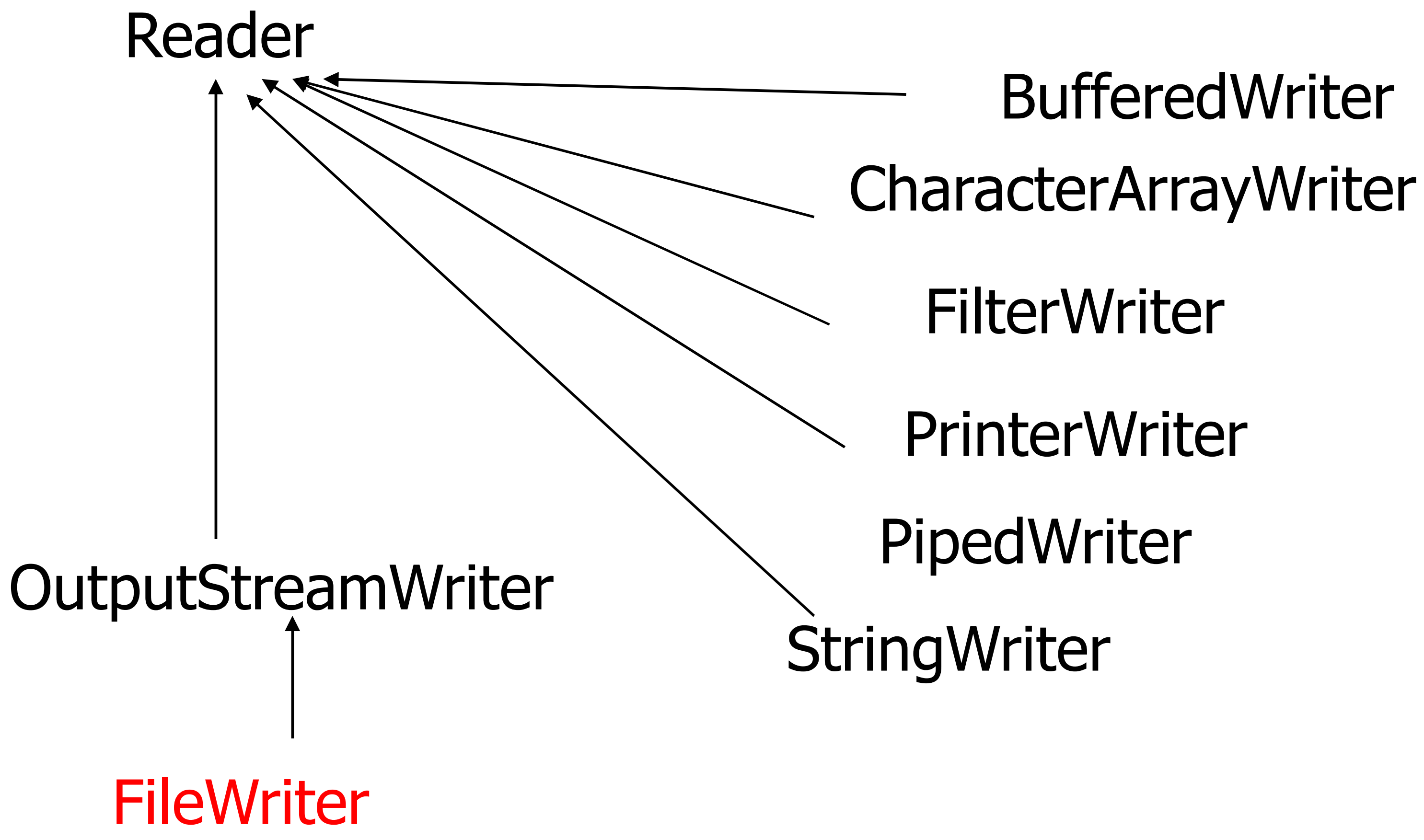
- Count total number of spaces in the file

```
import java.io.*;
public class CountSpace {
        public static void main (String[] args)
                throws IOException
        {
                Reader in;  // in can also be FileReader
                in = new FileReader("FileIn.txt");
                int ch, total, spaces;

                spaces = 0;

                for (total = 0 ; (ch = in.read()) != -1; total++){
                        if(Character.isWhitespace((char) ch))
                        {
                                spaces++;
                        }
                }
                System.out.println(total + " chars " + spaces + " spaces ");
        }
}
```

Writer Class Hierarchy



Byte Output Streams - operations

public abstract void write(int ch)	Write *ch* as characters.
public void write(char[] buf, int offset, int count)	Write *count* characters starting from *offset* in *buf.*
public void write(char[] buf)	Same as previous *offset=0* and *count = buf.length()*
public void write(String str, int offset, int count)	Write *count* characters starting at *offset* of *str.*
public void flush()	Flushes the stream.
public void close()	Closes stream

Copying Characters from Files

- Write a Program that copies contents of a source file to a destination file.
- The names of source and destination files is passed as command line arguments.
- Make sure that sufficient number of arguments are passed.
- Print appropriate error messages.

FileCopy.java

```
import java.io.*;
public class FileCopy {

	public static void main (String[] args)
	{
		if(args.length != 2)
		{
			System.out.println("Error: in sufficient arguments");
			System.out.println("Usage - java FileCopy SourceFile DestFile");
			System.exit(-1);
		}
		try {
		FileReader srcFile = new FileReader(args[0]);
		FileWriter destFile = new FileWriter(args[1]);

		int ch;
		while((ch=srcFile.read()) != -1)
			destFile.write(ch);
		srcFile.close();
		destFile.close();
		}
		catch(IOException e)
		{
			System.out.println(e);
			System.exit(-1);
		}
	}
}
```

Runs and Outputs

- Source file exists:
  - java FileCopy FileIn.txt Fileout.txt
- Source file does not exist:
  - java FileCopy abc Fileout.txt

    java.io.FileNotFoundException: abc (No such file or directory)
- In sufficient arguments passed
  - java FileCopy FileIn.txt

    Error: in sufficient arguments

    Usage - java FileCopy SourceFile DestFile

Buffered Streams

- Java supports creation of buffers to store temporarily data that read from or written to a stream. This process is known as *buffered I/O* operation.
- Buffered stream classes – BufferedInputStream, BufferedOutputStream, BufferedReader, BufferedWriter buffer data to avoid every read or write going to the stream.
- These are used in file operations since accessing the disk for every character read is not efficient.

Buffered Streams

- Buffered character streams understand lines of text.
- BufferedWriter has a newLine method which writes a new line character to the stream.
- BufferedReader has a readLine method to read a line of text as a String.
- For complete listing of methods, please see the Java manual/documentation.

BufferedReader - example

- Use a BufferedReader to read a file one line at a time and print the lines to standard output

```
import java.io.*;

class ReadTextFile {
        public static void main(String[] args)
                throws FileNotFoundException, IOException
        {
                BufferedReader in;
                in = new BufferedReader( new FileReader("Command.txt"));
                String line;
                while (( line = in.readLine())  !=  null )
              {
                  System.out.println(line);
              }
        }
}
```

Reading/Writing Bytes

- The FileReader and FileWriter classes are used to read and write 16-bit characters.
- As most file systems use only 8-bit bytes, Java supports number of classes that can handle bytes. The two most commonly used classes for handling bytes are:
  - FileInputStream (discussed earlier)
  - FileOutputStream

Writing Bytes - Example

```
public class WriteBytes {

        public static void main (String[] args)
        {
              byte cities[] = {'M', 'e', 'l', 'b', 'o', 'u', 'r', 'n', 'e', '\n', 'S', 'y','d', 'n', 'e', 'y', '\n` };

                FileOutputStream outFile;
                try{
                        outFile = new FileOutputStream("City.txt");
                        outFile.write(cities);
                        outFile.close();
                }
                catch(IOException e)
                {
                        System.out.println(e);
                        System.exit(-1);
                }

        }
}
```

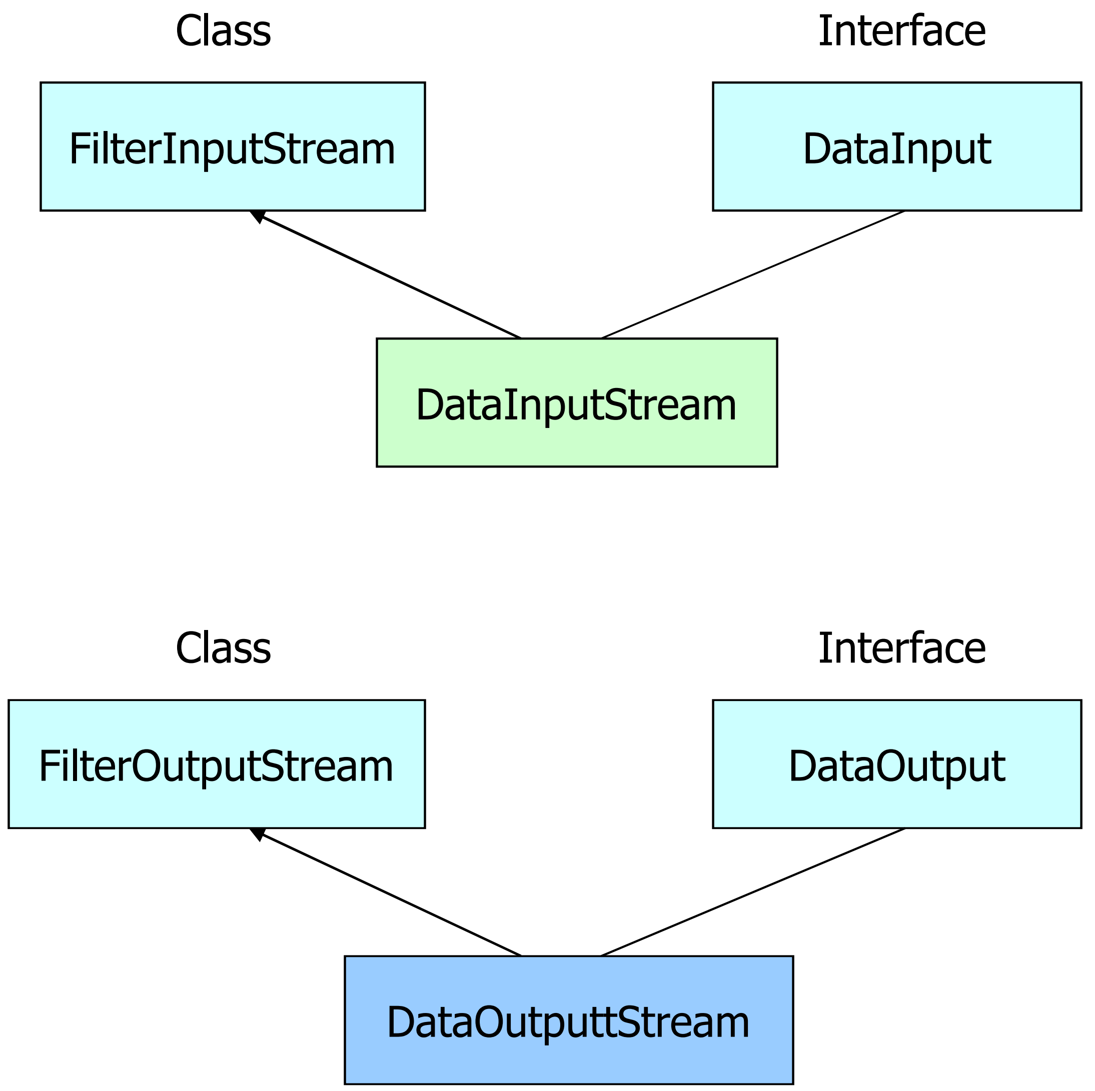
Summary

- All Java I/O classes are designed to operate with Exceptions.
- User Exceptions and your own handler with files to manger runtime errors.
- Subclasses FileReader / FileWriter support characters-based File I/O.
- FileInputStream and FileOutputStream classes support bytes-based File I/O.
- Buffered read operations support efficient I/O operations.

Handling Primitive Data Types

- The basic input and output streams provide read/write methods that can be used for reading/writing bytes or characters.
- To read/write the primitive data types such as integers and doubles, we can use filter classes as wrappers on the existing I/O streams to filter data to the original stream.
- The two filter classes supported for creating "data streams" for primitive data types are:
  - DataInputStream
  - DataOutputStream

Hierarchy of Data Stream Classes



Data Input Stream Creation

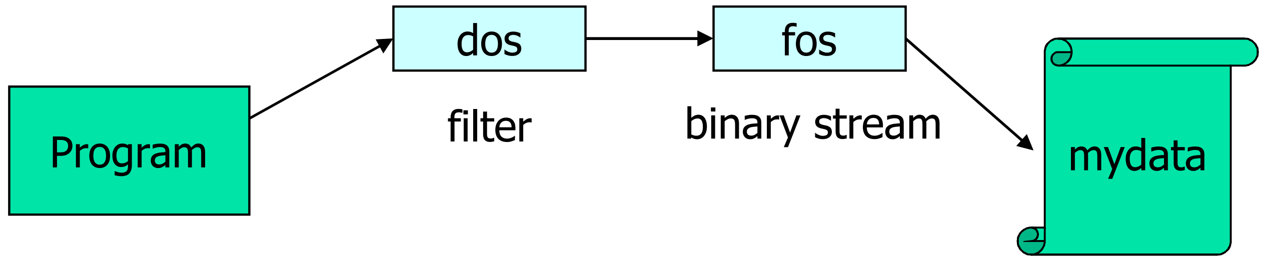
- Create Input File Stream:
  - FileInputStream fis = new FileInputStream(“InFile”);
- Create Input Data Stream:
  - DataInputStream dis = new DataInputStream(fis);
- The above statements wrap data input stream (dis) on file input stream (fis) and use it as a “filter”.
- Methods Supported:
  - readBoolean(), readByte(), readChar(), readShort(), readInt(), readLong(), readFloat(), readDouble()
- They read data stored in file in binary format.

Data Output Stream Creation

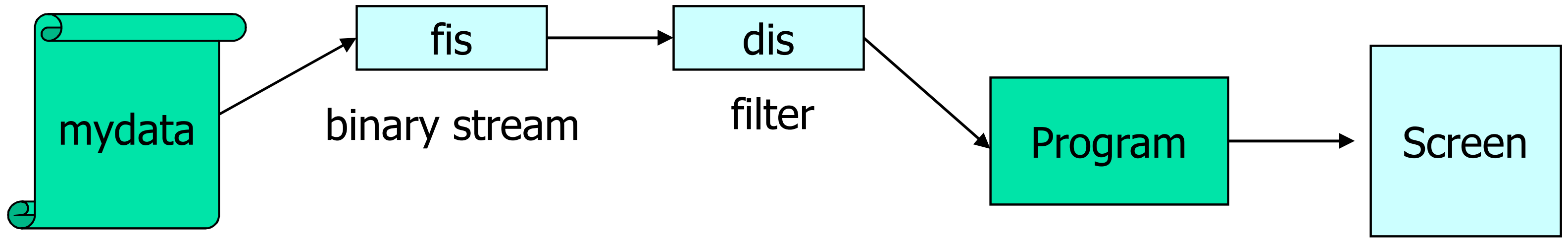
- Create Output File Stream:
  - FileOutputStream fos = new FileOutputStream(“OutFile”);
- Create Output Data Stream:
  - DataOutputStream dos = new DataOutputStream(fos);
- The above statements wrap data output stream (dos) on file output stream (fos) and use it as a “filter”.
- Methods Supported:
  - writeBoolean(), writeByte(), writeChar(), writeShort(), writeInt(), writeLong(), writeFloat(), writeDouble()
- They write data to file in binary format.
  - How many bytes are written to file when for statements:
    - writeInt(120), writeInt(10120)

Data Streams Flow via Filter

- Write primitive data to the file using a filter.



- Read primitive data from the file using a filter.



Writing and Reading Primitive Data

```
import java.io.*;
public class ReadWriteFilter {
    public static void main(String args[]) throws IOException {
        // write primitive data in binary format to the "mydata" file
        FileOutputStream fos = new FileOutputStream("mydata");
        DataOutputStream dos = new DataOutputStream(fos);
        dos.writeInt(120);
        dos.writeDouble(375.50);
        dos.writeInt('A'+1);
        dos.writeBoolean(true);
        dos.writeChar('X');
        dos.close();
        fos.close();

        // read primitive data in binary format from the "mydata" file
        FileInputStream fis = new FileInputStream("mydata");
        DataInputStream dis = new DataInputStream(fis);
        System.out.println(dis.readInt());
        System.out.println(dis.readDouble());
        System.out.println(dis.readInt());
        System.out.println(dis.readBoolean());
        System.out.println(dis.readChar());
        dis.close();
        fis.close();
    }
}
```

Program Run and Output

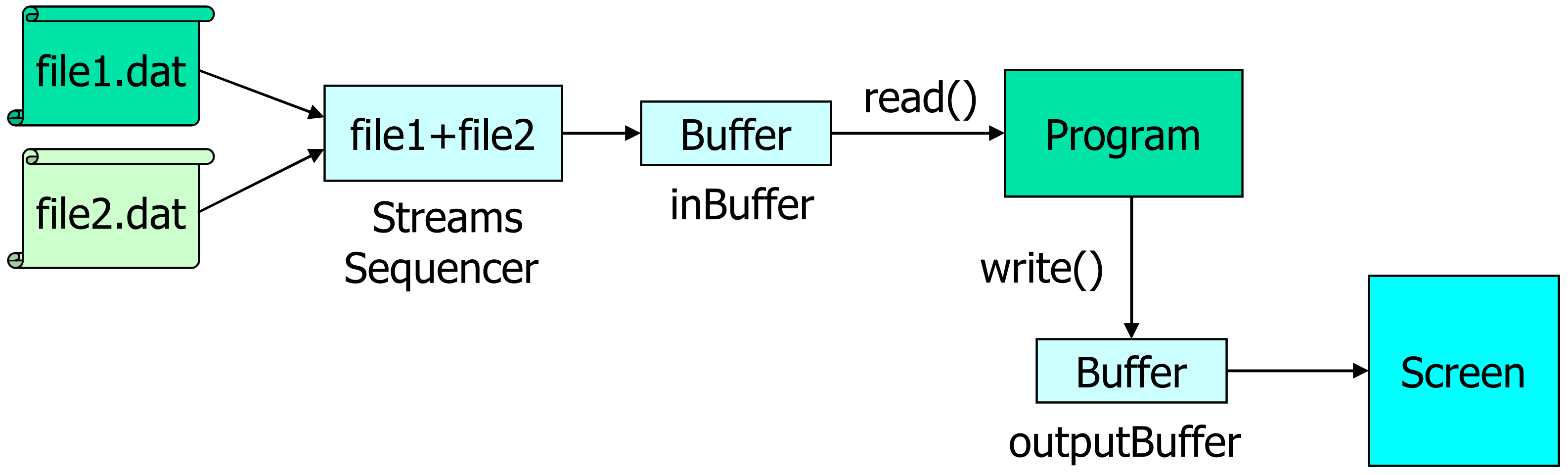
- C:\254\examples>java ReadWriteFilter
  - 120
  - 375.5
  - 66
  - true
  - X
- Display content of “mydata” file (in binary format):
  - C:\254\examples>type mydata
    - x@wx   B☺ X
- What is the size of “mydata” file (in bytes) ?
  - Size of int+double+int+boolean+char

Concatenating and Buffering Streams

- Two or more input streams can be combined into a single input stream. This process is known as logical *concatenation* of streams and is achieved using the *SequenceInputStream* class.
- A *SequenceInputStream* starts out with an ordered collection of input streams and reads from the first one until end of file is reached, whereupon it reads from the second one, and so on, until end of file is reached on the last of the contained input streams.

Sequencing and Buffering of Streams

- Buffered streams sit between the program and data source/destination and functions like a filter or support efficient I/O. Buffered can be created using BufferedInputStream and BufferedOutputStream classes.



Example Program

```
import java.io.*;
public class CombineStreams {
   public static void main(String args[]) throws IOException {
      // declare file streams
      FileInputStream file1 = new FileInputStream("file1.dat");
      FileInputStream file2 = new FileInputStream("file2.dat");
      // declare file3 to store combined streams
      SequenceInputStream file3 = null;
      // concatenate file1 and file2 streams into file3
      file3 = new SequenceInputStream(file1, file2);
      BufferedInputStream inBuffer = new BufferedInputStream(file3);
      BufferedOutputStream outBuffer = new BufferedOutputStream(System.out);
      // read and write combined streams until the end of buffers
      int ch;
      while((ch = inBuffer.read()) != -1 )
        outBuffer.write(ch);
      outBuffer.flush(); // check out the output by removing this line
      System.out.println("\nHello, This output is generated by CombineFiles.java program");
      inBuffer.close();
      outBuffer.close();
      file1.close();
      file2.close();
      file3.close();
   }
}
```

Contents of Input Files

- The file1.dat contains:
  - Hello,
  - I am C++, born in AT&T.
- The file2.dat contains:
  - Hello,
  - I am Java, born in Sun Microsystems!

Output

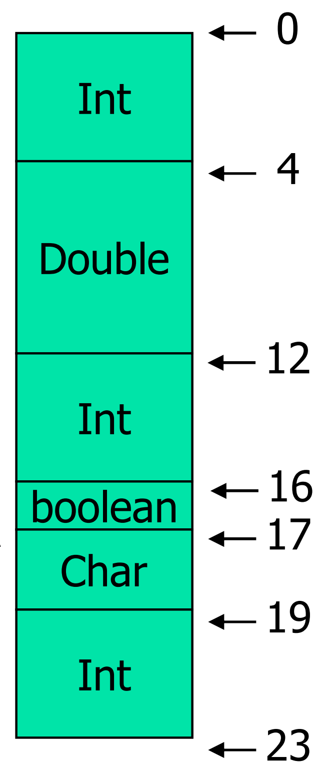
- C:\254\examples>java CombineStreams
  - Hello,
  - I am C++, born in AT&T.
  - Hello,
  - I am Java, born in Sun Microsystems!
  - Hello, This output is generated by CombineFiles.java program
- If the statement **outBuffer.flush()** is removed, the output will be:
  - Hello, This output is generated by CombineFiles.java program
  - Hello,
  - I am C++, born in AT&T.
  - Hello,
  - I am Java, born in Sun Microsystems!

Random Access Files

- So for we have discussed sequential files that are either used for storing data and accessed (read/write) them in sequence.
- In most real world applications, it is necessary to access data in non-sequential order (e.g, banking system) and append new data or update existing data.
- Java IO package supports RandomAccessFile class that allow us to create files that can be used for reading and/or writing with random access.
- The file can be open either in read mode (“r”) or read-write mode (“rw”) as follows:
  - myFileHandleName = new RandomAccessFile (“filename”, “mode”);
- The file pointer can be set to any to any location (measured in bytes) using seek() method prior to reading or writing.

Random Access Example

```
import java.io.*;
public class RandomAccess {
   public static void main(String args[]) throws IOException {
      // write primitive data in binary format to the "mydata" file
      RandomAccessFile myfile = new RandomAccessFile("rand.dat", "rw");
      myfile.writeInt(120);
      myfile.writeDouble(375.50);
      myfile.writeInt('A'+1);
      myfile.writeBoolean(true);
      myfile.writeChar('X');
     // set pointer to the beginning of file and read next two items
      myfile.seek(0);
      System.out.println(myfile.readInt());
      System.out.println(myfile.readDouble());
    //set pointer to the 4th item and read it
      myfile.seek(16);
      System.out.println(myfile.readBoolean());
      // Go to the end and “append” an integer 2003
      myfile.seek(myfile.length());
      myfile.writeInt(2003);
      // read 5th and 6th items
      myfile.seek(17);
      System.out.println(myfile.readChar());
      System.out.println(myfile.readInt());
      System.out.println("File length: "+myfile.length());
      myfile.close();
   }
}
```



Execution and Output

- C:\254\examples>java RandomAccess
  - 120
  - 375.5
  - true
  - X
  - 2003
  - File length: 23

Streams and Interactive I/O

- Real world applications are designed to support interactive and/or batch I/O operations.
- Interactive programs allow users to interact with them during their execution through I/O devices such as keyboard, mouse, display devices (text/graphical interface), media devices (microphones/speakers), etc..
  - Java provides rich functionality for developing interactive programs.
- Batch programs are those that are designed to read input data from files and produce outputs through files.

Standard I/O

- The System class contains three I/O objects (static)
  - System.in – instance of InputStream
  - System.out – instance of PrintStream
  - System.err – instance of PrintStream
- To perform keyboard input, we need use functionalities of DataInputStream and StringTokenizer classes.

Reading Integer from Standard Input

- Create buffered reader for standard input by wrapping System.in object:
  - BufferedReader dis = new BufferedReader(new InputStreamReader(System.in));
- Read a line of text from the console
  - String str = dis.readLine();
- Create Tokenens
  - StringTokenizer st;
  - st = new StringTokenizer(str);
- Convert String Token into basic integer:
  - int stdID = Integer.parseInt(st.nextToken());

Interactive IO Example

```
import java.io.*;
import java.util.*;
public class StudentRecord {
   public static void main(String args[]) throws IOException {
      // Create buffered reader for standard input
      BufferedReader dis = new BufferedReader(new InputStreamReader(System.in));
      StringTokenizer st;
      // reading data from console
      System.out.print("Enter Student ID: ");
      st = new StringTokenizer(dis.readLine());
      int stdID = Integer.parseInt(st.nextToken());
      System.out.print("Enter Student Name: ");
      String stdName = dis.readLine();
      System.out.print("Enter Student Marks: ");
      st = new StringTokenizer(dis.readLine());
      int stdMarks = Integer.parseInt(st.nextToken());
      // write to console
      System.out.println("Student details are:");
      System.out.println("ID: "+stdID);
      System.out.println("Name: "+stdName);
      System.out.println("Marks: "+stdMarks);
   }
}
```

Run and Output

- C:\254\examples>java StudentRecord
  - Enter Student ID: 2002010
  - Enter Student Name: Mary Baker
  - Enter Student Marks: 85
  - Student details are:
  - ID: 2002010
  - Name: Mary Baker
  - Marks: 85

Summary

- All Java I/O classes are designed to operate with Exceptions.
- User Exceptions and your own handler with files to manger runtime errors.
- Subclasses FileReader / FileWriter support characters-based File I/O.
- FileInputStream and FileOutputStream classes support bytes-based File I/O.
- Buffered read/write operations support efficient I/O.
- DataInputStream and DataOutputStream classes support rich I/O functionality.
- RandomAccessFile supports access to any data items in files in any order.