Input and Output

B.Tech. (IT), Sem-5,
Core Java Technology (CJT)

Dharmsinh Desai University
Prof. (Dr.) H B Prajapati

Topics

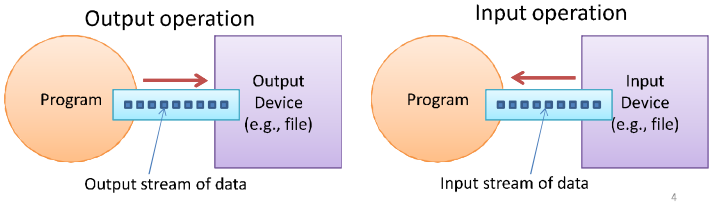
- Concepts of I/O in Java
  - Stream
  - Buffered I/O
  - Filtering
- Properties of a file or directory
- I/O classes
  - InputStream/OutputStream
  - Reader/Writer
- Binary I/O and Text I/O
- Reading from keyboard and writing to console
- Reading from file and writing to file
- Tokenizing input
- Use of Input/Output in GUI application

Introduction

- Input and Output is the basic requirement of any program/system.
- The java.io package deals with operations related to reading/writing to console, reading/writing to files, etc
- It contains many classes related to I/O operations for different types of read/write requirements.
- Java supports two kinds of I/O
  - Byte oriented
  - Character oriented (Unicode is of 2 bytes)

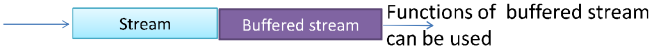
Stream

- Stream is a continuous flow of data
  - E.g., water flowing through a water pipe.
- A program can read from or write to an external entity (file, keyboard, network, etc.) through a stream



Stream based I/O

- A stream is linked to a physical device by the Java I/O system.
- A stream can be considered as a pipe through which data flows. The stream is an abstraction that either produces or consumes information
- All streams behave in same manner, even if the associated actual physical devices differ.
- Advantage of stream I/O is that one stream can be connected with another compatible stream.



Stream based I/O

- In Java, streams for both types of data have been defined: byte stream classes and character stream classes.
  - InputStream and OutputStream class hierarchies for byte oriented.
  - Reader and Writer class hierarchies for character oriented
- Character streams were not present in original version of Java (Java 1.0), but were added in Java 1.1.
- Character streams use Unicode.

- At the lowest level, all I/O is still byte-oriented.

File class

- It provides machine-independent abstraction for files, directories, and path names
- It describes the properties of a file itself, not the file content
  - permissions,
  - time, date,
  - directory path,
  - to navigate subdirectory hierarchies
- Deals with most of the machine-dependent complexities of files and path names
  - Windows: separator is \, but since it is a special character we need to write \\
  - Unix/Linux: separator is /. But it works on Windows also

File class and constructors

- public class java.io.File extends java.lang.Object implements java.io.Serializable, java.lang.Comparable
- Constuctors
  - File(String directoryPath)
  - File(String directoryPath, String filename)
  - File(File dirObj, String filename)
  - File(URI uriObj)

File class: methods

- public java.lang.String getName();
- public java.lang.String getParent();
- public java.io.File getParentFile();
- public java.lang.String getPath();
- public boolean isAbsolute();
- public java.lang.String getAbsolutePath();
- public java.io.File getAbsoluteFile();
- public java.lang.String getCanonicalPath() throws java.io.IOException;
- public java.io.File getCanonicalFile() throws java.io.IOException;

File class: methods

- public java.net.URL toURL() throws java.net.MalformedURLException;
- public java.net.URI toURI();
- public boolean canRead();
- public boolean canWrite();
- public boolean exists();
- public boolean isDirectory();
- public boolean isFile();
- public boolean isHidden();
- public long lastModified();
- public long length();

File class: methods

- public boolean createNewFile() throws java.io.IOException;
- public boolean delete();
- public void deleteOnExit();
- public java.lang.String[] list();
- public java.lang.String[] list(java.io.FilenameFilter);
- public java.io.File[] listFiles();
- public java.io.File[] listFiles(java.io.FilenameFilter);
- public java.io.File[] listFiles(java.io.FileFilter);
- public boolean mkdir();
- public boolean mkdirs();
- public boolean renameTo(java.io.File);

File class: methods

- public boolean setLastModified(long);
- public boolean setReadOnly();
- public boolean setWritable(boolean, boolean);
- public boolean setWritable(boolean);
- public boolean setReadable(boolean, boolean);
- public boolean setReadable(boolean);
- public boolean setExecutable(boolean, boolean);
- public boolean setExecutable(boolean);
- public boolean canExecute();

File class: methods

- public static java.io.File[] listRoots();
- public long getTotalSpace();
- public long getFreeSpace();
- public long getUsableSpace();
- public static java.io.File createTempFile(java.lang.String, java.lang.String, java.io.File) throws java.io.IOException;
- public static java.io.File createTempFile(java.lang.String, java.lang.String) throws java.io.IOException;
- public int compareTo(java.io.File);
- public boolean equals(java.lang.Object);

Program: FileProperties

```
import java.io.File;
import java.util.Date;
class FileProperties{
    public static void main(String[] args){
        File f1 = new File("FileProperties.java");
        File f2 = new File("P:\\reading materials\\Subject\\Core Java\\programs\\io\\FileProperties.java");
        System.out.println("File Name: " + f1.getName());
        System.out.println("Path: " + f1.getPath());
        System.out.println("Absolute Path: " + f1.getAbsolutePath());
        System.out.println("Parent: " + f1.getParent());
        System.out.println(f1.exists() ? "exists" : "does not exist");
```


Program: FileProperties

```
        System.out.println(f1.canWrite() ? "is writeable" : "is not writeable");
        System.out.println(f1.canRead() ? "is readable" : "is not readable");
        System.out.println("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
        System.out.println(f1.isFile() ? "is normal file" : "might be a named pipe");
        System.out.println(f1.isAbsolute() ? "is absolute" : "is not absolute");
```

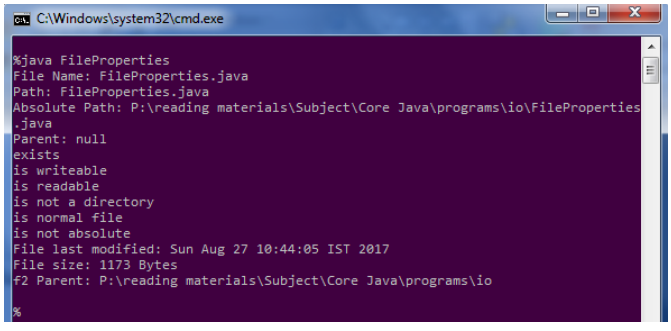

Program: FileProperties

```
        System.out.println("File last modified: " + new Date(f1.lastModified()));
        System.out.println("File size: " + f1.length() + " Bytes");

        System.out.println("f2 Parent: " + f2.getParent());
    }
}
```


Program: FileProperties

- Running the program



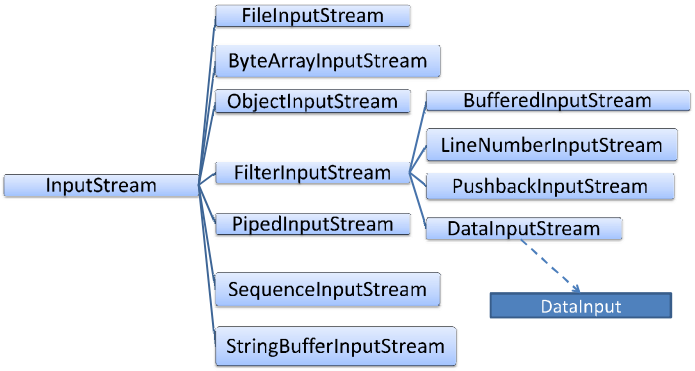
Stream classes

- InputStream and OutputStream are designed for byte streams.
- Reader and Writer are designed for character streams.
- The byte stream classes and the character stream classes form separate hierarchies.
- Which one to use?
  - We should use the character stream classes when working with characters or strings,
  - We should use the byte stream classes when working with bytes or other binary objects

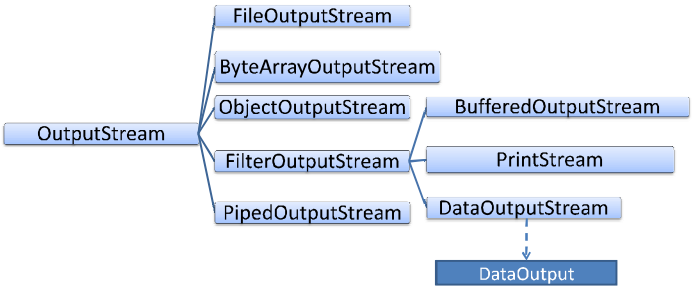
Predefined Streams

- System.in
  - Standard input, keyboard by default
  - It is an object of type InputStream
- System.out
  - Standard output stream, console by default
  - It is an object of type PrintStream
- System.err
  - Standard error, console by default
  - It is an object of type PrintStream

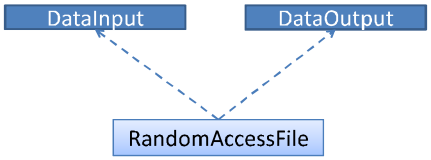
Class hierarchy: InputStream



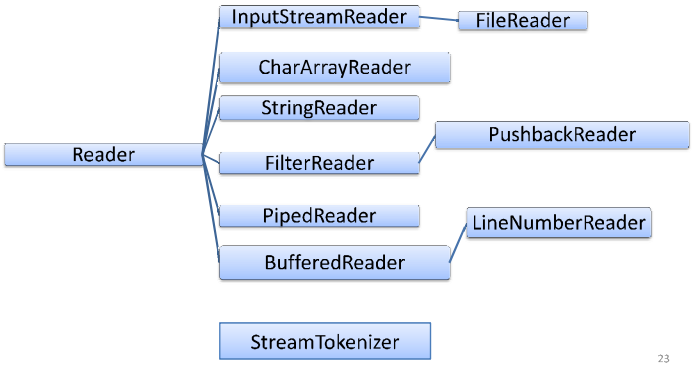
Class hierarchy: OutputStream



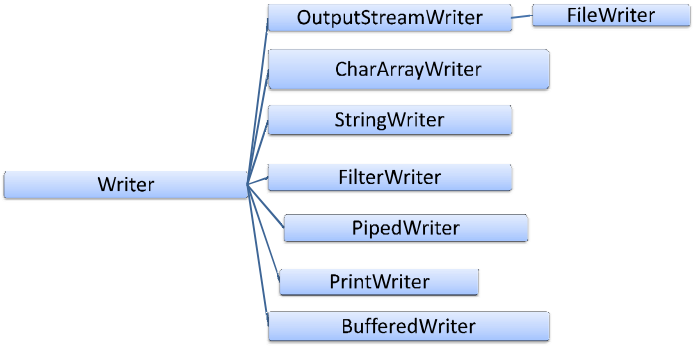
RandomAccessFile



Class hierarchy: Reader



Class hierarchy: Writer



InputStream and OutputStream

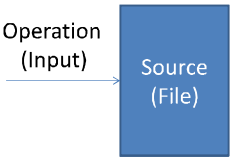
- InputStream is an abstract class that defines Java's model of streaming byte input. It implements the Closeable interface.
- OutputStream is an abstract class that defines streaming byte output. It implements the Closeable and Flushable interfaces.
- Most of the methods of these two classes throw IOException.

Reader and Writer

- Reader is an abstract class that defines Java's model of streaming character input. It implements the Closeable and Readable interface.
- Writer is an abstract class that defines streaming character output. It implements the Closeable, Appendable, and Flushable interfaces.
- Most of the methods of these two classes throw IOException.

Names of classes

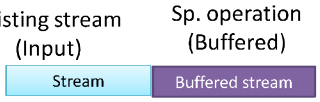
- Based on source or destination of data
  - File
  - ByteArray
  - CharArray
  - String
  - StringBuffer
  - Object



Name of class: FileInputStream

Names of classes

- Based on the way input or output operations are performed on data
  - Filter
  - Piped
  - Buffered
  - LineNumber
  - Pushback
  - Data
- Specialized functions
  - StreamTokenizer
  - RandomAccessFile



Name of class: BufferedInputStream

InputStream and Reader

- InputStream is an abstract class that defines streaming byte input
- It implements Closeable interface and all of its methods can throw IOException.
  - public abstract int read() throws IOException
  - public abstract int read(byte b[ ]) throws IOException
  - public abstract void close() throws IOException
  - public abstract int available () throws IOException
    - It is available only in InputStream not in Reader
  - public long skip(long bytes) throws IOException
- Both read methods are blocking, in no data is available.
  - It returns how many bytes are read
  - Returns -1 when there is end of data (EOF)

OutputStream and Writer

- OutputStream is an abstract class that defines streaming byte output
- It implements Closeable and Flushable interfaces and all of its methods return void and can throw IOException.
- As there is no return value (return type is void), status of method call is determined based on whether exception is thrown or not.
  - public abstract void write(int b) throws IOException
  - public abstract void write(byte[ ] buff) throws IOException
  - public abstract void close() throws IOException
  - public abstract void flush() throws IOException
    - It clears the buffer, i.e., sends the data to the stream.

Processing External Files

- Reading input from a file
  - FileInputStream
    - FileInputStream(String filePath)
    - FileInputStream(File fileObj)
  - Both can throw FileNotFoundException
  - Methods: mark() and reset() are not overridden by this class so its use will result in IOException

Processing External Files

- Writing to a file
  - FileOutputStream
    - FileOutputStream(String filePath)
    - FileOutputStream(File fileObj)
    - FileOutputStream(String filePath, boolean append)
    - FileOutputStream(File fileObj, boolean append)
  - Constructors can throw FileNotFoundException or SecurityException
  - If append is true, the file is opened in append mode.

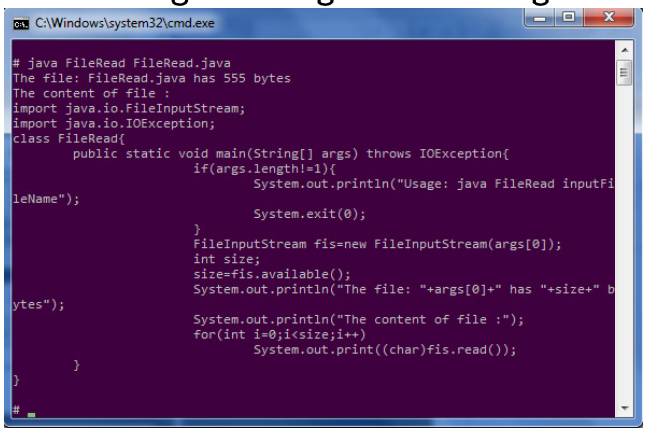
Program: Reading a file

```
import java.io.FileInputStream;
import java.io.IOException;
class FileRead{
    public static void main(String[] args) throws IOException{
        if(args.length!=1){
            System.out.println("Usage: java FileRead
inputFileName");
            System.exit(0);
        }
```


Program: Reading a file (cont…)

```
        FileInputStream fis=new FileInputStream(args[0]);
        int size;
        size=fis.available();
        System.out.println("The file: "+args[0]+" has
"+size+" bytes");
        System.out.println("The content of file :");
        for(int i=0;i<size;i++)
            System.out.print((char)fis.read());
    }
}
```


Running the Program: Reading a file



Program: Reading a file (without typecast to char)

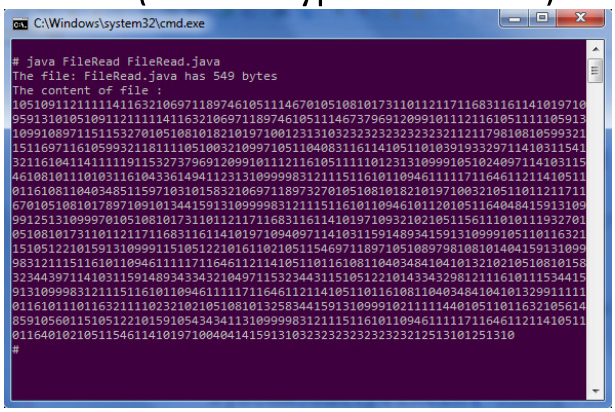
- Type casting to char is important

```
for(int i=0;i<size;i++)
    System.out.print((char)fis.read());
```

- If we do not typecast, then what will happen?

```
for(int i=0;i<size;i++)
    System.out.print(fis.read());
```

Program: Reading a file (without typecast to char)



Program: Using seek() to jump file read pointer

```
import java.io.FileInputStream;
import java.io.IOException;
class FilePortionRead{
	public static void main(String[] args) throws IOException{
		if(args.length!=1){
			System.out.println("Usage: java FileRead
  inputFileName");
			System.exit(0);
		}
		FileInputStream fis=new FileInputStream(args[0]);
		int size;
```

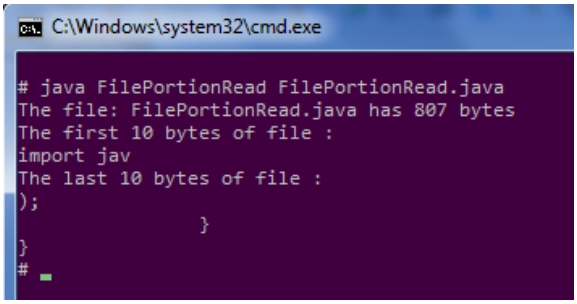

Program: Using seek() to jump file read pointer (cont…)

```
		size=fis.available();
		byte[] buffer=new byte[10];
		System.out.println("The file: "+args[0]+" has
"+size+" bytes");
		System.out.println("The first 10 bytes of file :");
		fis.read(buffer);
		for(int i=0;i<10;i++)
			System.out.print((char)buffer[i]);
		System.out.println();
```


Program: Using seek() to jump file read pointer (cont…)

```
		System.out.println("The last 10 bytes of file :");
		fis.skip(size-10-10);
		fis.read(buffer);
		for(int i=0;i<10;i++)
			System.out.print((char)buffer[i]);
	}
}
```


Running the Program: Using seek()



Program: file copy command

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.File;
import java.io.IOException;
import java.io.FileNotFoundException;
class FileCopy{
	public static void main(String[] args){
		FileInputStream fis=null;
		FileOutputStream fos=null;
		byte readByte;
```

Program: file copy command (cont…)

```
        if(args.length!=2){
                System.out.println("Usage: java FileCopy
SourceFileName DestFileName");
                System.exit(0);
        }
        try{
                fis=new FileInputStream(new
File(args[0]));
                //create file output stream if the file does
not exist
                File outFile=new File(args[1]);
```


Program: file copy command (cont…)

```
                if(outFile.exists()){
                        System.out.println("File:
"+args[1]+" already exists");
                        return;
                }
                else
                        fos=new
FileOutputStream(args[1]);
                do{
                        readByte = (byte) fis.read();
                        fos.write(readByte);
                }while(readByte!=-1);
        }
```

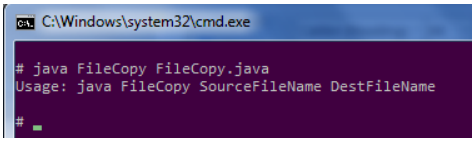

Program: file copy command (cont…)

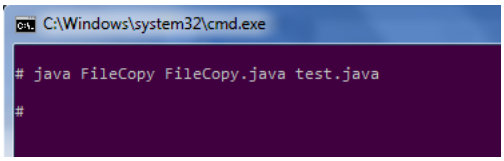
```
        catch(FileNotFoundException e){
                System.out.println("File: "+args[0]+" not
found");
        }
        catch(IOException e){
                System.out.println(e.getMessage());
        }
        finally{
                try{
                        if(fis!=null) fis.close();
                        if(fos!=null) fos.close();
                }
```


Program: file copy command (cont…)

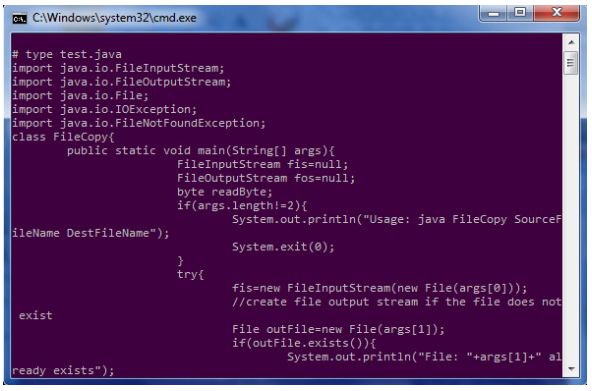
```
                catch(IOException e){}
        }
    }
}
```


Running the Program: file copy





Running the Program: file copy



Program: writing using character stream

```
import java.io.IOException;
import java.io.File;

class CharacterOrientedWrite{
    public static void main(String[] args)throws IOException{
            FileWriter fw=null;
            if(args.length!=1){
                System.out.println("Usage: java
  CharacterOrientedWrite  textFileName.txt");
                System.exit(0);
            }
```

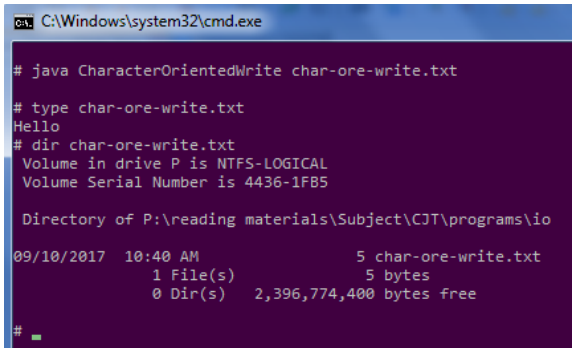

Program: writing using character stream (cont…)

```
            File file=new File(args[0]);
            if(file.exists()){
                System.out.println("The file: "+args[0]+
                    " already exists,"+
                    "delete it, and rerun the
program");
                System.exit(0);
            }
```


Program: writing using character stream (cont…)

```
            //Writing data as text
            fw=new FileWriter(file);
            fw.write("Hello");
            if(fw!=null)
                fw.close();
    }
}
```


Running Program: writing using character stream



Array Streams

- In FileInputStream (and FileReader) and FileOutputStream (and FileWriter) are source and destination of input and output operation, respectively.
- Similarly, Java allows ByteArray and CharArray as source and destination of input and output operations.
  - Input
    - ByteArrayInputStream
    - CharArrayReader
  - Output
    - ByteArrayOutputStream
    - CharArrayWriter

Array Streams

- What is the use of these streams?
  - ByteArrayInputStream implements both mark() and reset().
  - We can read same input multiple times.
  - E.g., while transferring a file content from one machine to another in network
    - We can read content from a file and write it to a byte buffer (i.e., array) on source machine
    - Then transfer the buffer at once to another machine.

Array Streams

- Constructors
  - ByteArrayInputStream(byte[ ] inputByteArray)
  - ByteArrayInputStream(byte[ ] inputByteArray, int start, int numBytes)
  - CharArrayReader(char[ ] inputCharArray)
  - CharArrayReader(char[ ] inputCharArray, int start, int numChars)
  - ByteArrayOutputStream()
  - ByteArrayOutputStream(int numBytes)
  - CharArrayWriter()
  - CharArrayWriter(int numChars)
- When we use default constructors for ByteArrayOutputStream and CharArrayWriter, default buffer size is 32 bytes.
- The buffer size is increased automatically if needed.

Program: read byte array twice

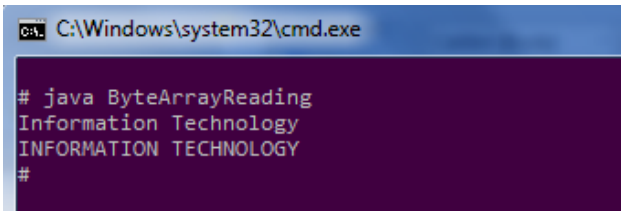
```
import java.io.ByteArrayInputStream;
import java.io.IOException;
class ByteArrayReading{
    public static void main(String[] args) throws IOException{
        String inputString="Information Technology";
        byte[] b=inputString.getBytes();
        ByteArrayInputStream bais=new
ByteArrayInputStream(b);
        int c;
```


Program: read byte array twice (cont…)

```
        while((c=bais.read())!=-1){
            System.out.print((char)c);
        }
        bais.reset();
        System.out.println();
        while((c=bais.read())!=-1){

    System.out.print(Character.toUpperCase((char)c));
        }
    }
}
```


Running the Program: read byte array twice



Filter Streams

- Basic input or output stream provides methods to read or write byte or characters.
- However, filter streams can filter bytes or characters for some purposes.
  - E.g., instead of reading individual byte or character, we may want to read integer, double, or float.
- We should use FilterInputStream and FilterOutputStream to filter bytes.
- We should use BufferedReader and PushbackReader to filter characters when we need to process strings.

Filter Streams

- FilterInputStream and FilterOutputStream are abstract classes. We use their subclasses for our purposes.
- Subclasses of FilterInputStream

Class	Usage
DataInputStream	Allows reading of all primitive data
BufferedInputStream	Reads data from the buffer, reads/loads data from the underlying stream if needed
LineNumberInputStream	Keeps track of how many lines are read
PushbackInputStream	Allows single byte look-ahead. After reading a byte, it can be pushed back to the stream

Filter Streams

- Subclasses of FilterOutputStream

Class	Usage
DataOutputStream	Allows writing of all primitive data in binary formats
BufferedOutputStream	Outputs to the buffer first and then outputs to the stream, if required. We can use flush() method to write the content of the buffer to the actual stream
PrintStream	Allows writing of all primitive types in unicode format.

Reading Console Input

- In Java 1.0, console input was possible with use of byte stream, as character stream was not present.
- Though, byte stream can be used to read console input, it is not recommended.
- Rather, we should use character-oriented stream for console input.
- We can read from console using System.in, which is byte stream oriented.

Reading Console Input

- BufferReader supports a buffered input stream, for which constructor is
  - BufferedReader(Reader inputReader)
- InputStreamReader can convert bytes into characters. Constructor is
  - InputStreamReader(InputStream inputStream)
- Thus, keyboard input is performed using the following code

```
BufferedReader br=new BufferedReader(new
    InputStreamReader(System.in));
String inputLine=br.readLine();
```

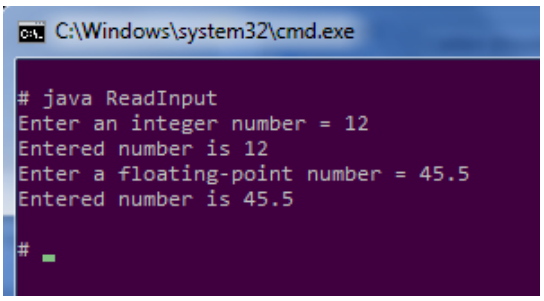

Program: reading console input

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
class ReadInput{
    public static void main(String[] args) throws IOException{
            BufferedReader br=new BufferedReader(new
  InputStreamReader(System.in));
            System.out.print("Enter an integer number = ");
            int i=Integer.parseInt(br.readLine().trim());
            System.out.println("Entered number is "+i);
```


Program: reading console input (cont…)

```
            System.out.print("Enter a floating-point number =
");
            float f= Float.parseFloat(br.readLine().trim());
            System.out.println("Entered number is "+f);
        }
}
```


Running the Program: reading console input



PrintStream & PrintWriter

- Use of System.out to write to the console is recommended mostly for debugging purposes or for sample programs.
- For real-world programs, use of PrintWriter stream is recommended.
- Constructor
  - PrintWriter(OutputStream outputStream, boolean flushOnNewline)
  - If flushOnNewline is true, flushing automatically takes place. If false, flushing is not automatic
- PrintWriter supports the print() and println() methods for all types including Object.
  - If an argument is not a simple type (i.e., an object), the PrintWriter methods call the object's toString() method.

PrintStream & PrintWriter

- To write to the console by using a PrintWriter, specify System.out for the output stream and flush the stream after each newline.
- PrintWriter pw = new PrintWriter(System.out, true);

Program: writing data as text

```
import java.io.PrintWriter;
import java.io.FileOutputStream;
import java.io.File;
class TextWrite{
    public static void main(String[] args)throws IOException{
            PrintWriter pw=null;
            if(args.length!=1){
                System.out.println("Usage: java TextWrite
  textFileName.txt");
                System.exit(0);
            }
```

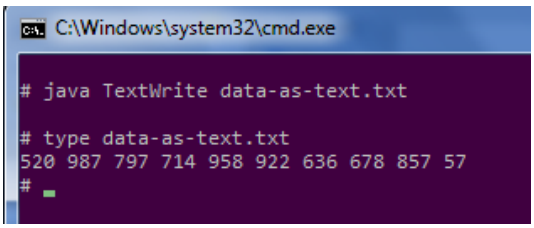

Program: writing data as text (cont…)

```
            File file=new File(args[0]);
            if(file.exists()){
                System.out.println("The file: "+args[0]+
                    " already exists,"+
                    "delete it, and rerun the
program");
                System.exit(0);
            }
```


Program: writing data as text (cont…)

```
            //Writing data as text
            pw=new PrintWriter(new FileOutputStream(file));
            for(int i=0;i<10;i++)
                pw.print((int)(Math.random()*1000)+" ");
            if(pw!=null)
                pw.close();
    }
}
```


Running Program: writing data as text



PrintStream & PrintWriter: formatted output

- Formatting was added in J2SE 1.5
- Useful methods in PrintStream class
  - PrintStream printf(String fmtString, Object[] args)
  - PrintStream printf(String fmtString, Object... args)
  - PrintStream format(String fmtString, Object[] args)
  - PrintStream format(String fmtString, Object... args)
- Useful methods in PrintWriter class
  - PrintWriter printf(String fmtString, Object[] args)
  - PrintWriter printf(String fmtString, Object... args)
  - PrintWriter format(String fmtString, Object[] args)
  - PrintWriter format(String fmtString, Object... args)

PrintStream & PrintWriter: formats

- %d
  - integer number
- %+d
  - display + in front of positive number, to align
- %(d
  - display negative as bracketed, instead of negative
- % d
  - One space between % and d: to align with negative numbers
- %5d
  - Print integer number in the specified with, right justified
- %05d
  - Print integer number in the specified with, with leading zeros

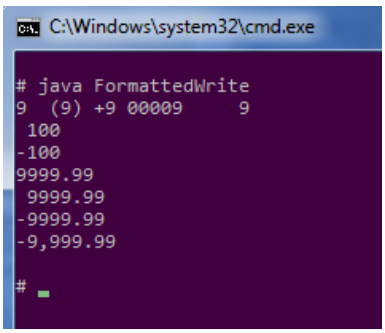
PrintStream & PrintWriter: formats

- %f
  - Floating point number
- %.2f
  - Two digits after decimal point
- % .2f
  - One space after %: to align positive number with negative number
- %,.2f
  - Format number with comma (,)

Program: displaying formatted data

```
class FormattedWrite{
    public static void main(String[] args){
            //Formatting integer data
        System.out.printf("%d  %(d %+d %05d %5d\n",9,-9,9,9,9);
            System.out.printf("% d\n",100);
            System.out.printf("%+d\n",-100);
            //Formatting floating-point data
            System.out.printf("%.2f\n",9999.99);
            System.out.printf("% .2f\n",9999.99);
            System.out.printf("% .2f\n",-9999.99);
            System.out.printf("%,.2f\n",-9999.99);
    }
}
```


Running the Program: displaying formatted data



Data Streams

- The data streams (DataInputStream and DataOutputStream) read and write Java primitive types in a machine-independent (little-endian or big-endian) fashion.
  - It is possible to write binary file on one machine and read it on another machine having different OS or platform.
- DataInputStream extends FilterInputStream and implements DataInput interface
- DataOutputStream extends FilterOutputStream and implements DataOutput interface
- DataInputStream and DataOutputStream are generally used to avail facility specified by DataInput and DataOutput interfaces.

Methods of DataInput

- public byte readByte() throws IOException
- public short readShort() throws IOException
- public int readInt() throws IOException
- public long readLong() throws IOException
- public float readFloat() throws IOException
- public double readDouble() throws IOException
- public char readChar() throws IOException
- public boolean readBoolean() throws IOException
- public String readLine() throws IOException
- public String readUTF() throws IOException

Methods of DataOutput

- public void writeByte(int b) throws IOException
- public void writeShort(int s) throws IOException
- public void writeChar(int c) throws IOException
- public void writeInt(int i) throws IOException
- public void writeLong(long l) throws IOException
- public void writeFloat(float f) throws IOException
- public void writeDouble(double d) throws IOException
- public void writeBytes(String s) throws IOException
- public void writeChars(String s) throws IOException
- public void writeUTF(String s) throws IOException

How data streams are used?

- The data streams are often used as wrappers on existing input or output streams (e.g., files) to filter data in original stream.
  - Data Input
    - public DataInputStream(InputStream inStream)
  - Data output
    - public DataOutputStream(OutputStream outStream)
- Examples

DataInputStream inFile=new DataInputStream(new FileInputStream("in.data"))

DataOutputStream outFile=new DataOutputStream(new FileOutputStream("out.data"))

Program: binary read-write

```
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.FileNotFoundException;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.File;
class BinaryReadWrite{
    public static void main(String[] args)throws IOException{
        DataInputStream dis=null;
        DataOutputStream dos=null;
```


Program: binary read-write (cont…)

```
        if(args.length!=1){
            System.out.println("Usage: java
BinaryReadWrite dataFileName.dat");
            System.exit(0);
        }
        File file=new File(args[0]);
        if(file.exists()){
            System.out.println("The file: "+args[0]+
                " already exists,"+
                "delete it, and rerun the
program");
            System.exit(0);
        }
```


Program: binary read-write (cont…)

```
        //Writing data
        try{
            dos=new DataOutputStream(new
FileOutputStream(file));
            for(int i=0;i<10;i++)

    dos.writeInt((int)(Math.random()*1000));
        }
        catch(IOException e){
            System.out.println(e.getMessage());
        }
```

Program: binary read-write (cont…)

```
        finally{
            try{
                if(dos!=null)
                    dos.close();
            }
            catch(IOException e){}
        }
```

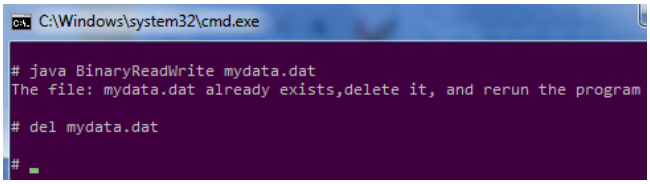

Program: binary read-write (cont…)

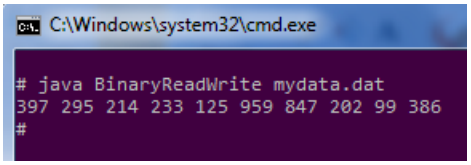
```
        //Reading data
        try{
            dis=new DataInputStream(new
FileInputStream(file));
            for(int i=0;i<10;i++)
                System.out.print(dis.readInt()+" ");
        }
        catch(FileNotFoundException e){
            System.out.println(e.getMessage());
        }
        catch(IOException e){
            System.out.println(e.getMessage());
        }
```


Program: binary read-write (cont…)

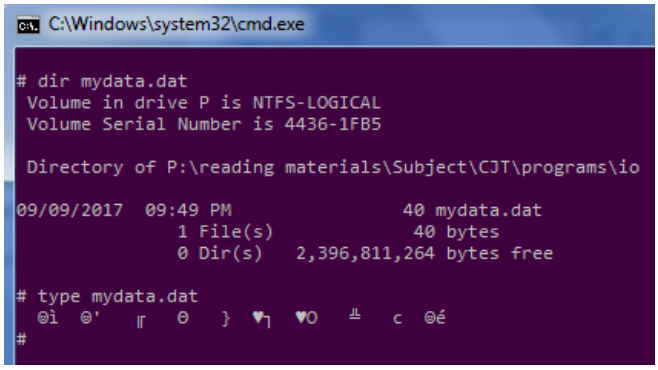
```
        finally{
            try{
                if(dos!=null)
                    dos.close();
            }
            catch(IOException e){}
        }
    }
}
```


Running the Program: binary read-write





Running the Program: binary read-write



Stream Tokenizer: Parsing Text files

- StreamTokenizer class provides facility of tokenizing a file
- StreamTokenizer generates tokens (separated words) and allows to get one at a time.
- StreamTokenizer class has three variables
  - int ttype
    - Indicates current token type
  - double nval
    - Gives the value of current token if it is a number(integer or floating point)
  - String sval
    - Gives the string value if the token is not a number

Stream Tokenizer: Parsing Text files

- StreamTokenizer constants
  - public static final int TT_EOF;
  - public static final int TT_EOL;
  - public static final int TT_NUMBER;
  - public static final int TT_WORD;
- Method to go to the next token
  - public int nextToken()
- How to use?

```
if(st.ttype==st.TT_NUMBER)
   double value=st.nval;
```

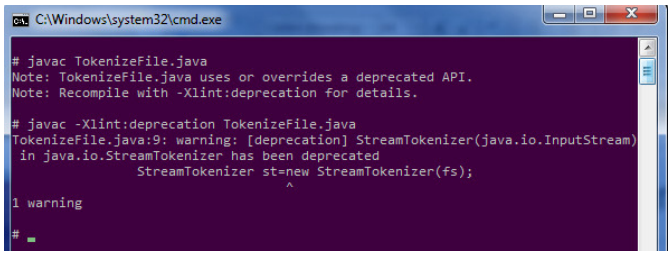

Program: parsing a text file

```
import java.io.*;
class TokenizeFile{
    public static void main(String[] args)throws Exception{
        if(args.length!=1){
                System.out.println("Usage: java TextWrite
    textFileName.txt");
                System.exit(0);
        }
        FileInputStream fs=new FileInputStream(args[0]);
        StreamTokenizer st=new StreamTokenizer(fs);
```


Program: parsing a text file (cont…)

```
        int t;
        double sum=0.0;;
        while((t=st.nextToken())!=st.TT_EOF){
                if(t==st.TT_NUMBER){
                        System.out.println(st.nval+" ");
                        sum+=st.nval;
                }
                else if(t==st.TT_WORD)
                    System.out.println(st.sval+" is not a number");
        }
        System.out.println("Sum of above numbers is "+sum);
    }
}
```

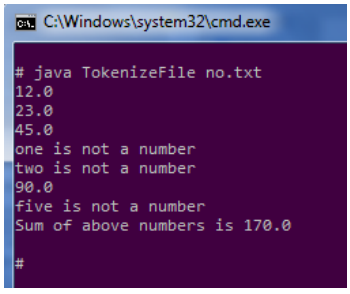

Compiling Program: parsing a text file

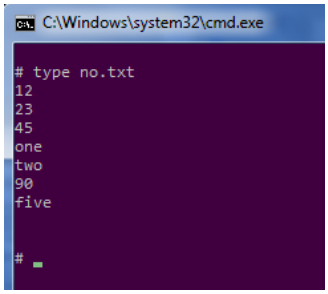


- Constructor taking InputStream is deprecated
- We can use the following

```
FileReader fr=new FileReader(args[0]);
StreamTokenizer st=new StreamTokenizer(fr);
```


Running Program: parsing a text file





Random Access File

- All of the streams used so far are either read-only or write-only streams.
- Those streams are sequential
  - Every time there is change in data, we need to recreate the file and need to rewrite the whole data.
  - RandomAccessFile supports positioning requests (we can position the file pointer within the file)
- RandomAccessFile class allows both reading and writing
  - It is not derived from InputStream or OutputStream
  - It implements DataInput and DataOutput interfaces, and it also implements Closeable interface.

Random Access File

- Constructors
  - RandomAccessFile(File fileObj, String access) throws FileNotFoundException
  - RandomAccessFile(String fileName, String access) throws FileNotFoundException
- Examples
  - new RandomAccessFile("result.data", "rw");

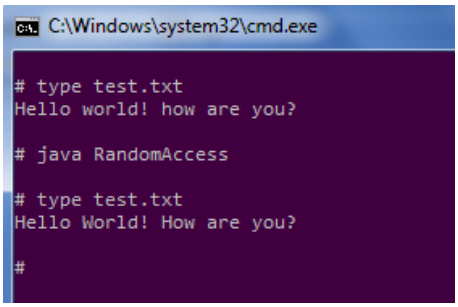
Random Access File

- Useful methods
  - void seek(long newPos) throws IOException
  - void skipBytes(int skip) throws IOException
    - Skips specified number of bytes relative to the current position
  - long getFilePointer() throws IOException
  - long length() IOException
  - void writeChar(int v) throws IOException
    - Writes character as two byte Unicode
  - void writeUTF(String s) throws IOException
  - void setLength(long length) throws IOException
    - Use to shorten or lengthen the file

Program: random access

```
import java.io.RandomAccessFile;
import java.io.IOException;
class RandomAccess{
    public static void main(String[] args)throws IOException{
        RandomAccessFile raf=new
    RandomAccessFile("test.txt","rw");
        raf.seek(5);
        raf.writeChar('W');
        raf.skipBytes(5);
        raf.writeChar('H');
    }
}
```


Running Program: random access



Program: Use in GUI application, notepad

```
import java.io.FileInputStream;
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.awt.Frame;
import java.awt.TextField;
import java.awt.TextArea;
import java.awt.Panel;
import java.awt.Button;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
```


Program: Use in GUI application, notepad (cont…)

```
class Notepad extends Frame implements ActionListener{
    TextArea ta;
    TextField tf;
    Button b;
    public static void main(String[] args){
        Frame f=new Notepad();
    }
    public Notepad(){
        super("Notepad");
        ta=new TextArea();
        tf=new TextField(35);
        b=new Button("Show");
```

Program: Use in GUI application, notepad (cont…)

```
Panel p=new Panel();
p.add(tf);
p.add(b);
b.addActionListener(this);
add(ta);
add("South",p);
setSize(800,600);
setVisible(true);
tf.requestFocus();
setVisible(true);
}
```


Program: Use in GUI application, notepad (cont…)

```
public void actionPerformed(ActionEvent ae){
    FileInputStream fs=null;
    BufferedReader br=null;
    String s=ae.getActionCommand();
    String str=null;
    if(s.equals("Show")){
        try{
            fs=new
FileInputStream(tf.getText().trim());
            br=new BufferedReader(new
InputStreamReader(fs));
            boolean first=true;
```


Program: Use in GUI application, notepad (cont…)

```
            while((str=br.readLine())!=null){
                if(first){
                    ta.append(str);
                    first=false;
                }
                else{
                    ta.append("\n"+str);
                }
            }
            ta.setCaretPosition(0);
            ta.requestFocus();
        }
```

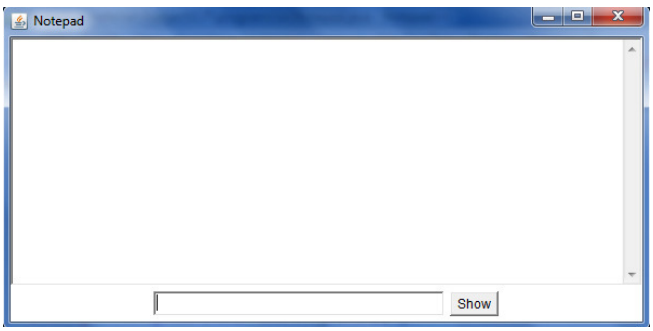

Program: Use in GUI application, notepad (cont…)

```
        catch(Exception oe){
            System.out.println(oe);
        }
        finally{
            if(fs!=null){
                try{
                    fs.close();
                }
                catch(Exception e){
                    System.out.println(e);
                }
            }
        }
```


Program: Use in GUI application, notepad (cont…)

```
    }
}
}
```


Running Program: Use in GUI application, notepad



Running Program: Use in GUI application, notepad