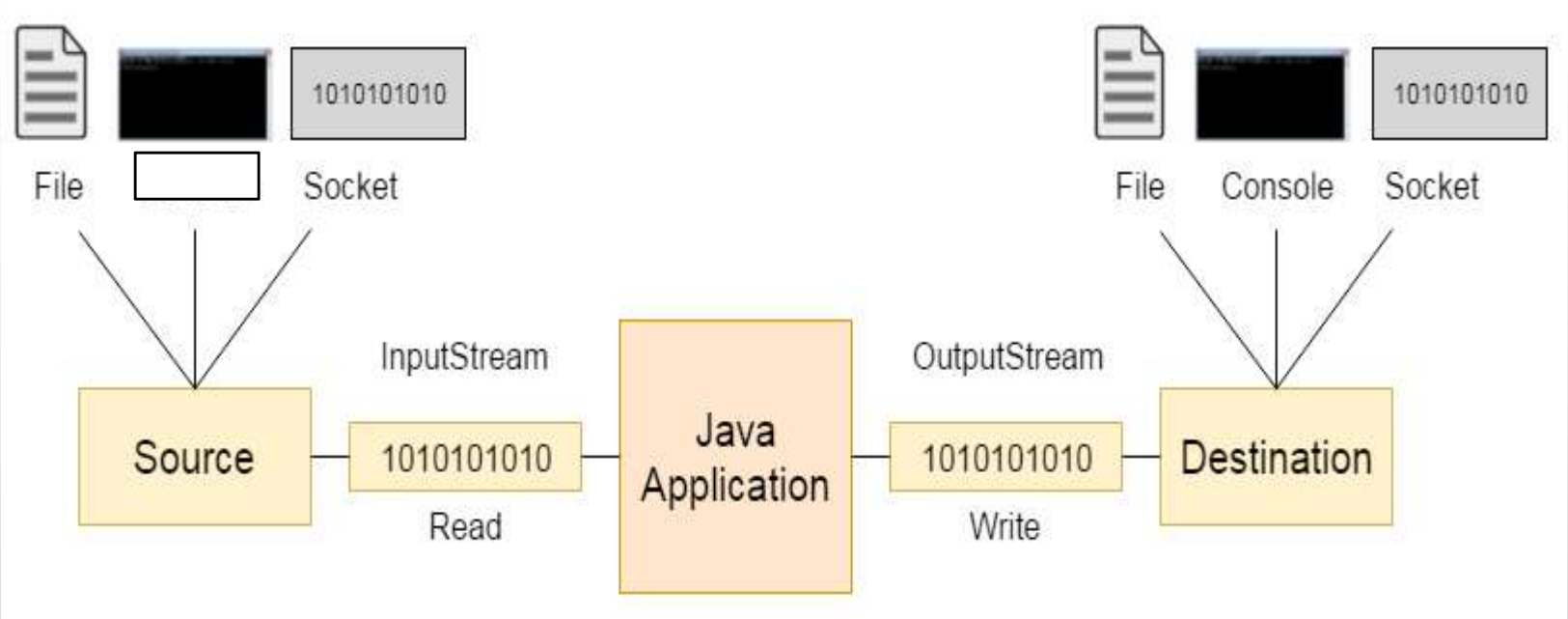
IO Streams

PresentationPoint


1010101010
File
Socket
1010101010
File
Console
Socket
InputStream
OutputStream
Source
1010101010
Read
Java Application
1010101010
Write
Destination

- There are many classes and interfaces in the **java.io** package.

File

- A **File** object is used to obtain or manipulate the information associated with a disk file.

- Ex: permissions, time, date, and directory path etc.

File

- **File** class does not specify how information is retrieved from or stored in files;

- It describes the properties of a file itself.

File

- A directory in Java is treated simply as a **File** with one additional property:

- List of filenames that can be examined by the **list( )** method.

Constructors used to create **File** objects:

File(String *directoryPath*)

File(String *directoryPath*, String *filename*)

File(File *dirObj*, String *filename*)

File(URI *uriObj*)

Constructors used to create **File** objects:

```
File f1 = new File(“c:/mit/ict/cce/prg1.txt");
```

Constructors used to create **File** objects:

```
File f2 = new File(" c:/mit/ict/cce/", “prg1.txt”);
```

Constructors used to create **File** objects:

```
File f3 = new File(" c:/mit/ict/cce/");
File f4 = new File(f3,”prg1.txt”);
```

Windows:

recognize forward slash(/).

If we use backaword slash, we need to use escape sequence(\\)

Unix :

recognize backword slash(\).

Methods :

String getName()

Methods :

String getParent()

Methods :

String getPath()

Methods :

String getAbsolutePath()

Methods :

boolean exists()

Methods :

boolean canWrite()

boolean canRead()

Methods :

boolean isDirectory()

boolean isFile()

Methods :

boolean isAbsolute()

Methods :

long lastModified()

long length()

Methods :

boolean renameTo(File *newName*)

- *newName* becomes the new name of the invoking **File** object.

- It will return **true** upon success and **false** if the file cannot be renamed.

(if you either attempt to rename a file so that it uses an existing filename).

Creating Directories

boolean mkdir( )

boolean mkdirs( )

mkdirs : To create a directory for which no path exists. It creates a directory and all the parents of the directory.

Creating Directories

```
File f = new File("D:\\MIT");

if (f.mkdir())
{
    System.out.println("Directory is created");
}
else
 {
       System.out.println("Directory cannot be created");
 }
```

Boolean delete()

To delete file and directory.

To delete directory, it should be empty

Directories

String [] list()

The listFiles( ) Alternative

- Variation to the **list( )** method.
- Which returns File instead of string

File[ ] listFiles( )

File[ ] listFiles(FilenameFilter *FFObj*)

Using FilenameFilter

- To limit the number of files returned by the **list( )** method.
- To include only those files that match a certain filename pattern, or *filter.*
- Second form of **list( )** :

String[ ] list(FilenameFilter *FFObj*)

- FFObj is an object of a class that implements the **FilenameFilter** interface.

- **FilenameFilter** defines a single method, **accept( )**, which is called once for each file in a list.

boolean accept(File *directory*, String *filename*)

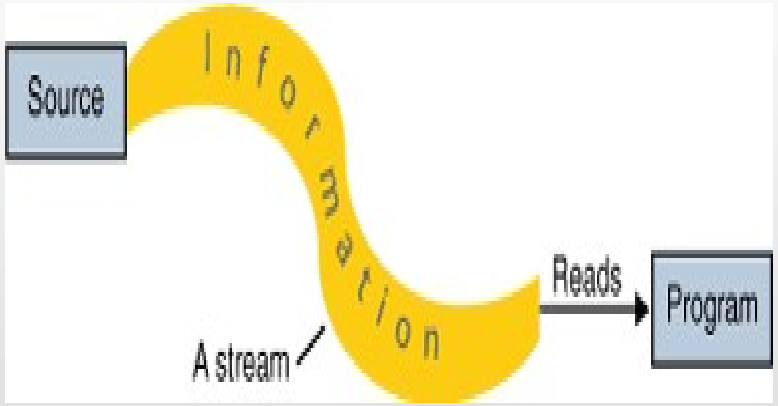
- The **accept( )** method returns **true** for files in the directory specified by *directory* that should be included in the list (that is, those that match the *filename* argument), and returns **false** for those files that should be excluded.

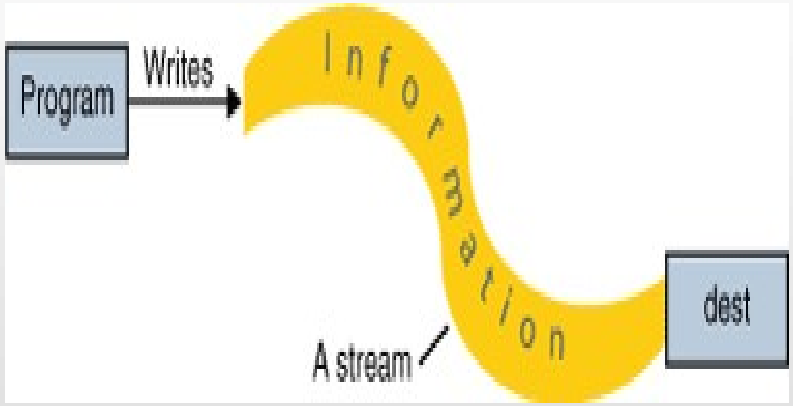
- The **OnlyExt** class implements **FilenameFilter**.

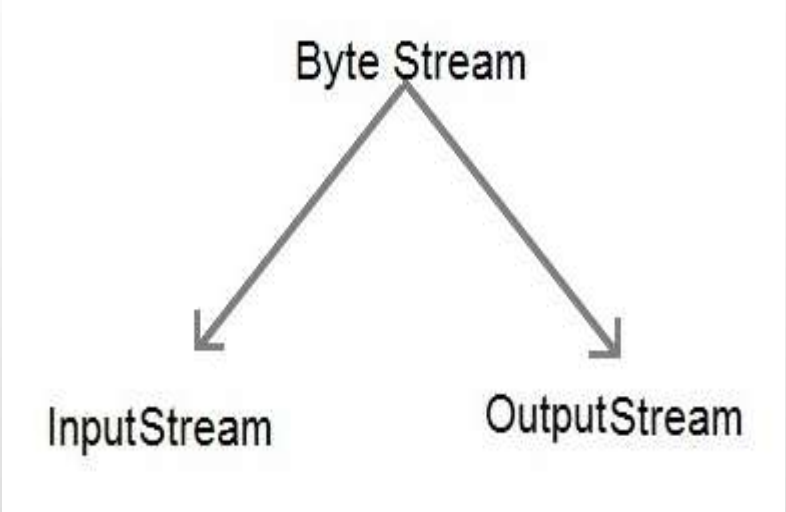
```
import java.io.*;
public class OnlyExt implements FilenameFilter
{
        String ext;
        public OnlyExt(String ext)
         {
                this.ext = "." + ext;
         }
        public boolean accept(File dir, String name)
        {
                return name.endsWith(ext);
        }
}
```

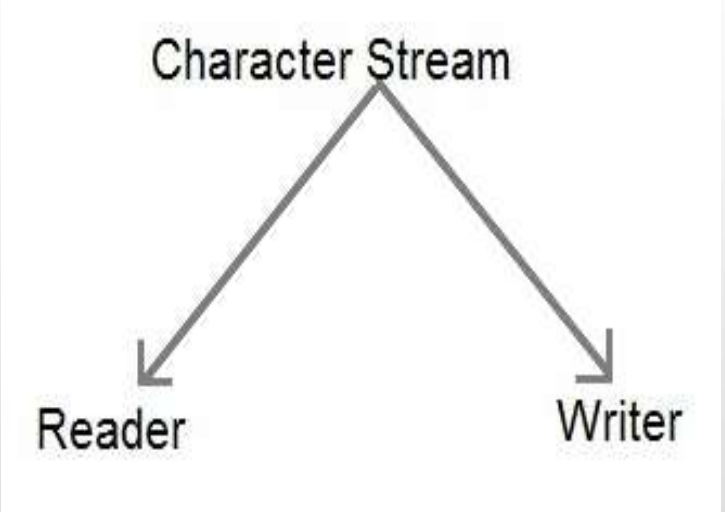
```
import java.io.*;
class DirListOnly
 {          public static void main(String args[])
           {
                    String dirname = "/java";
                    File f1 = new File(dirname);
                    FilenameFilter only = new OnlyExt("html");
                    String s[] = f1.list(only);
                    for (int i=0; i < s.length; i++)
                    {    System.out.println(s[i]);
                    }
           } }
```

- A stream is a logical entity that either produces or consumes information.
- A stream is linked to a physical device by the Java I/O system.






Byte Stream
InputStream
OutputStream


Character Stream
Reader
Writer

The Stream Classes

- Java's stream-based I/O is built upon four abstract classes:

  InputStream,

  OutputStream,

  Reader,

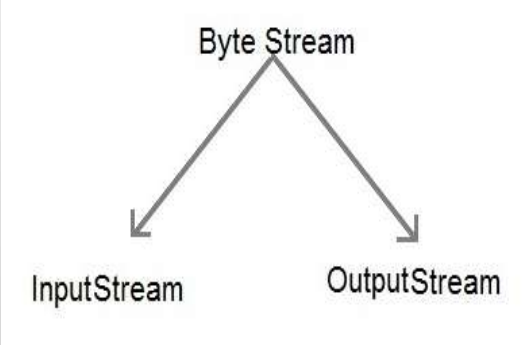
  Writer.

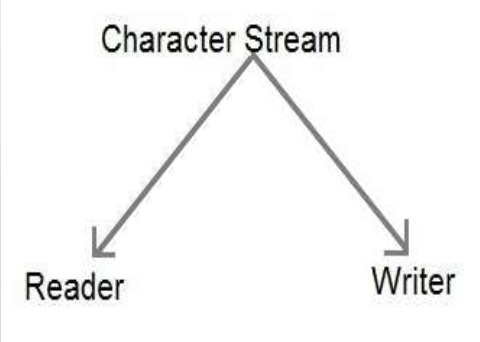
- Used to create several concrete stream subclasses.

- Programs perform their I/O operations through subclasses.

- The top-level classes define the basic functionality common to all stream classes.

- **InputStream** and **OutputStream** are designed for byte streams.
- **Reader** and **Writer** are designed for character streams.
- **use the character stream classes when working with characters or strings, and use the byte stream classes when working with bytes or other binary objects.**

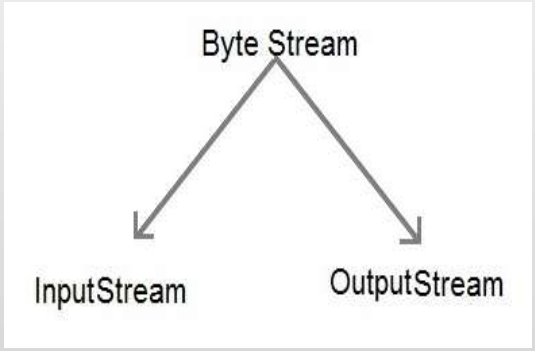




The Byte Streams

- Byte stream classes provide a rich environment for handling byte-oriented I/O.

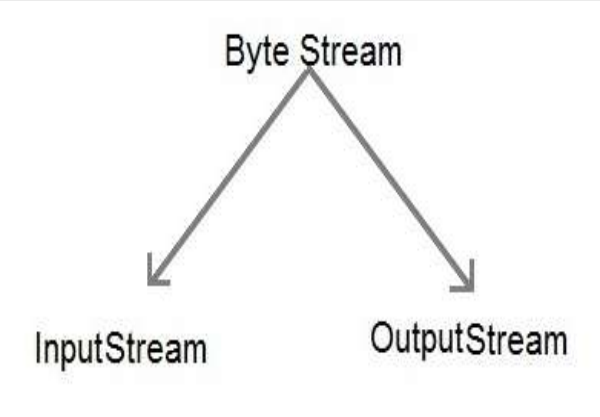
- A byte stream can be used with any type of object, including binary data.



InputStream

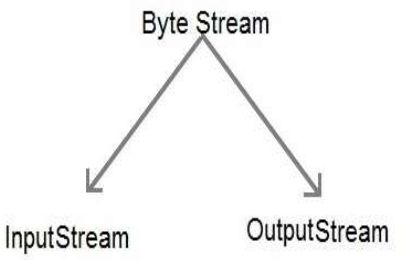
- **InputStream** is an abstract class that defines Java's model of streaming byte input.

- Most of the methods in this class will throw an **IOException** on error conditions.



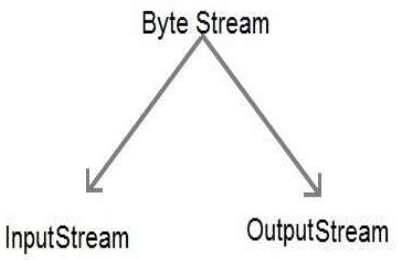
methods in **InputStream**.

int read( )



- Returns an integer representation of the next available byte of input.

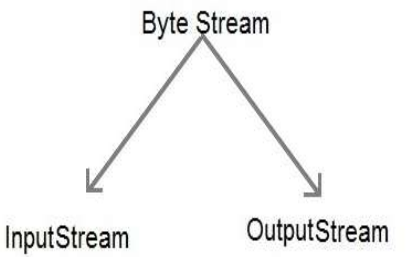
methods in **InputStream**.



int read(byte buffer[ ])

- Attempts to read up to buffer.length bytes into buffer and returns the actual number of bytes that were successfully read.

methods in **InputStream**.



int read(byte buffer[ ], int offset, int numBytes)

methods in **InputStream**.

int available( )

- Returns the number of bytes of input currently available for reading.

methods in **InputStream**.

int available( )

long skip(long numBytes)

- Ignores (that is, skips) numBytes bytes of input, returning the number of bytes actually ignored.

methods in **InputStream**.

int available( )

long skip(long numBytes)

void close( )

- Closes the input source. Further read attempts will generate an IOException.

methods in **InputStream**.

boolean markSupported( )

void mark(int numBytes)

void reset( )

- Returns true if mark( )/reset( ) are supported by the invoking stream.
- Places a mark at the current point in the input stream that will remain valid until numBytes bytes are read.
- Resets the input pointer to the previously set mark.

Table 19-1 shows the methods in **InputStream.**

Method	Description
int available( )	Returns the number of bytes of input currently available for reading.
void close( )	Closes the input source. Further read attempts will generate an **IOException**.
void mark(int *numBytes*)	Places a mark at the current point in the input stream that will remain valid until *numBytes* bytes are read.
boolean markSupported( )	Returns **true** if **mark( )/reset( )** are supported by the invoking stream.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ])	Attempts to read up to *buffer.length* bytes into *buffer* and returns the actual number of bytes that were successfully read. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long *numBytes*)	Ignores (that is, skips) *numBytes* bytes of input, returning the number of bytes actually ignored.

OutputStream

- **OutputStream** is an abstract class that defines streaming byte output.

- Most of the methods in this class return **void** and throw an **IOException** in the case of errors.

methods in **OutputStream**.

void write(int b)

Writes a single byte to an output stream.

methods in **OutputStream**.

void write(byte buffer[ ])

Writes a complete array of bytes to an output stream.

methods in **OutputStream**.

void write(byte buffer[ ], int offset, int numBytes)

methods in **OutputStream**.

void close( )

void flush( )

Closes the output stream. Further write attempts will generate an IOException.

Finalizes the output state so that any buffers are cleared.

Table 19-2 shows the methods in **OutputStream**.

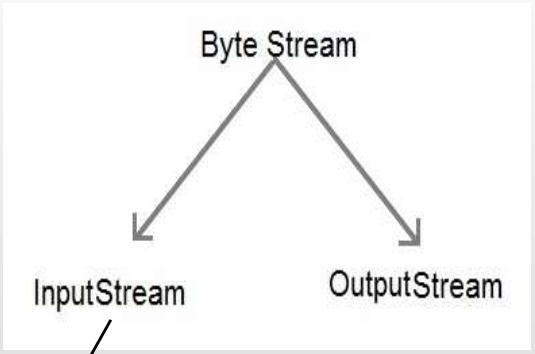
Method	Description
void close( )	Closes the output stream. Further write attempts will generate an **IOException**.
void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(int *b*)	Writes a single byte to an output stream. Note that the parameter is an **int**, which allows you to call **write( )** with expressions without having to cast them back to **byte**.
void write(byte *buffer*[ ])	Writes a complete array of bytes to an output stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].

FileInputStream

- The **FileInputStream** class creates an **InputStream** that we can use to read bytes from a file.

FileInputStream(String *filepath*)

FileInputStream(File *fileObj*)



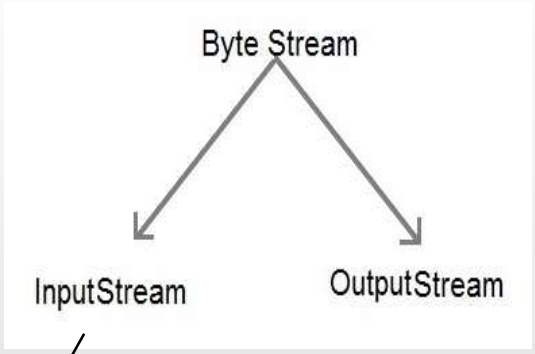
- Either can throw a **FileNotFoundException**.

FileInputStream

Example

```
FileInputStream f0 = new FileInputStream (“c:/mit/ict/cce/prg1.txt");
```

```
File f = new File (“c:/mit/ict/cce/prg1.txt");
FileInputStream f1 = new FileInputStream(f);
```



FileInputStream

```
 import java.io.*;
public static void main(String args[]) throws IOException
{       int size;
    InputStream f =  new FileInputStream("FileInputStreamDemo.java");
    System.out.println("Total Available Bytes: " +  (size = f.available()));
    int n = size/40;
    System.out.println("Reading first " + n + " bytes of the file");
     for (int i=0; i < n; i++)
    {
                System.out.print((char) f.read());
   }
```

```
Total Available Bytes: 1433
First 35 bytes of the file one read() at a time
// Demonstrate FileInputStream.
im
```

```
System.out.println("\nStill Available: " + f.available());
System.out.println("Reading the next " + n +"  bytes");
byte b[] = new byte[n];
if (f.read(b) != n)
{
    System.err.println("couldn't read " + n + " bytes.");
}
System.out.println(new String(b, 0, n));
```

```
Still Available: 1398
Reading the next 35 with one read(b[])
port java.io.*;

class FileInputS
```

```
 System.out.println("\nStill Available: " + (size = f.available()));
 System.out.println("Skipping half of remaining bytes with skip()");
 f.skip(size/2);
 System.out.println("Still Available: " + f.available());
 System.out.println("Reading " + n/2 + " into the end of array");
 if (f.read(b, n/2, n/2) != n/2)
 {
        System.err.println("couldn't read " + n/2 + " bytes.");
 }
  System.out.println(new String(b, 0, b.length));
  System.out.println("\nStill Available: " + f.available());
  f.close();
 }
}
```

```
Still Available: 1363
Skipping half of remaining bytes with skip()
Still Available: 682
Reading 17 into the end of array
port java.io.*;
read(b) != n) {
S

Still Available: 665
```

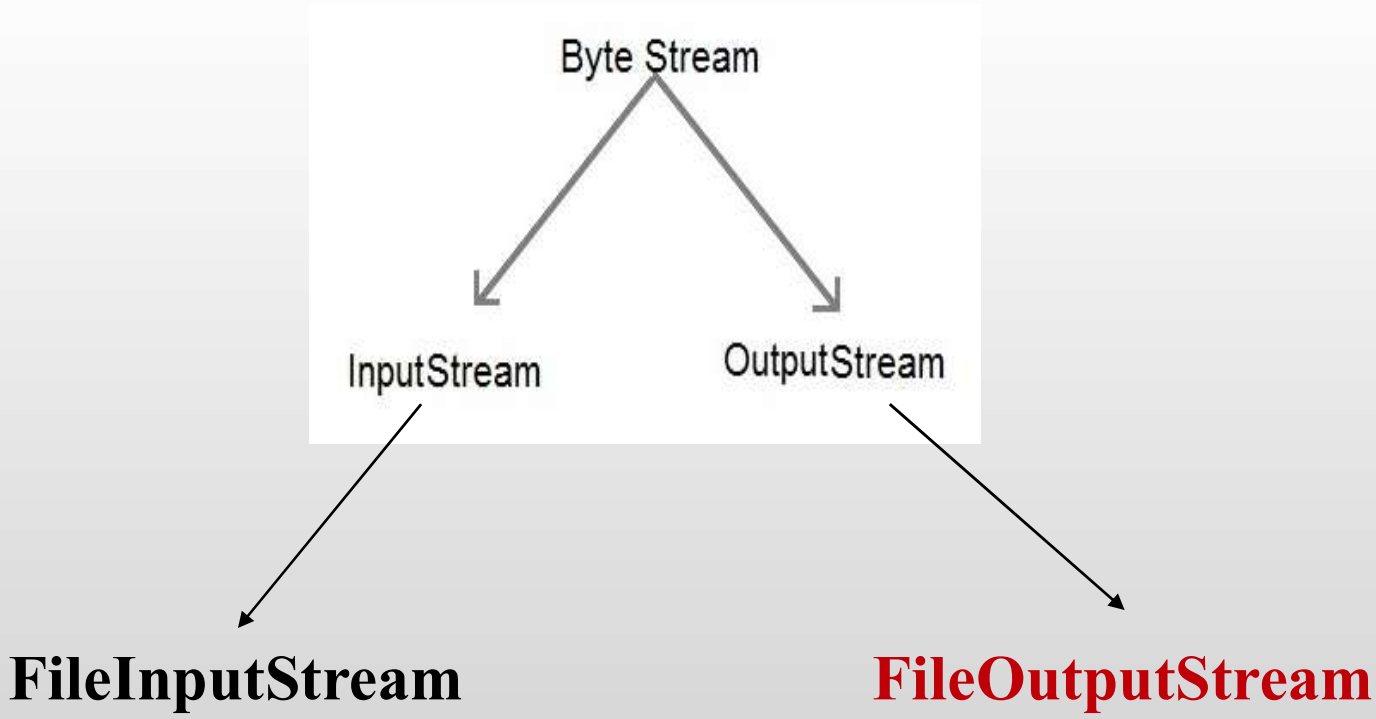
NOTE

- The preceding example handle any I/O exceptions that might occur by throwing **IOException** out of **main( )**, which means that they are handled by the JVM.

- This is fine for simple demonstration programs, but commercial applications will normally need to handle I/O exceptions within the program.

FileOutputStream

- **FileOutputStream** creates an **OutputStream** that you can use to write bytes to a file.



FileOutputStream

FileOutputStream(String *filePath*)

FileOutputStream(File *fileObj*)

FileOutputStream(String *filePath*, boolean *append*)

FileOutputStream(File *fileObj*, boolean *append*)

- They can throw a **FileNotFoundException**.
- If *append* is **true**, the file is opened in append mode.
- Creation of a **FileOutputStream** is not dependent on the file already existing.
- **FileOutputStream** will create the file before opening it for output when we create the object.
- In the case where you attempt to open a read-only file, an **IOException** will be thrown.

```
// Demonstrate FileOutputStream.
import java.io.*;
class FileOutputStreamDemo
{       public static void main(String args[]) throws IOException
        {
            String source = "Now is the time for all good men\n"
                                + " to come to the aid of their country\n"
                                + " and pay their due taxes.";
            byte buf[] = source.getBytes();
            OutputStream f0 = new FileOutputStream("file1.txt");
            for (int i=0; i < buf.length; i += 2)
            {               f0.write(buf[i]);
            }
        f0.close();
```

```
        OutputStream f1 = new FileOutputStream("file2.txt");
        f1.write(buf);
        f1.close();

        OutputStream f2 = new FileOutputStream("file3.txt");
        f2.write(buf,buf.length-buf.length/4,buf.length/4);
        f2.close();
    }
}
```

- The first, **file1.txt**, will contain every other byte from the sample.
- The second, **file2.txt**, will contain the entire set of bytes.
- The third and last, **file3.txt**, will contain only the last quarter.

```
String source = "Now is the time for all good men\n"
                + " to come to the aid of their country\n"
                + " and pay their due taxes.";
```

```
Nwi h iefralgo e
t oet h i ftercuty n a hi u ae.
```

Next, **file2.txt**:

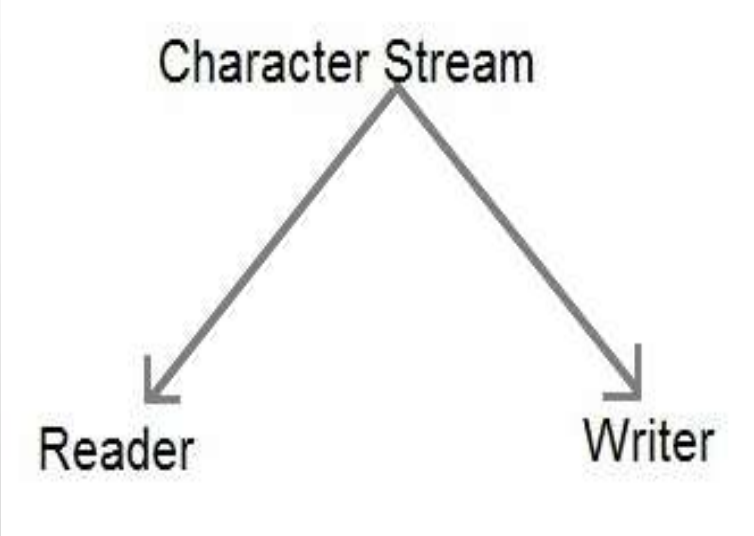
```
Now is the time for all good men
 to come to the aid of their country
 and pay their due taxes.
```

Finally, **file3.txt**:

```
nd pay their due taxes.
```

The Character Streams

- Top of the character stream hierarchies are the **Reader** and **Writer** abstract classes.



The Character Streams

- The byte stream classes provide functionality to handle any type of I/O operation, they cannot work directly with Unicode characters.

- Character streams : direct I/O support for characters.

NOTE :

- **T**he character I/O classes were added by the 1.1 release of Java.
- Because of this, we may still find legacy code that uses byte streams where character streams would be more appropriate

Reader

- **Reader** is an abstract class that defines Java's model of streaming character input.

Reader methods

int read( )

Reader methods

int read(char buffer[ ])

Reader methods

int read(char buffer[ ], int offset, int numChars)

Reader methods

long skip(long numBytes)

void close( )

Reader methods

boolean markSupported( )

void mark(int numChars)

void reset( )

Table 19-3 provides a synopsis of the methods in **Reader**.

Method	Description
abstract void close( )	Closes the input source. Further read attempts will generate an **IOException**.
void mark(int *numChars*)	Places a mark at the current point in the input stream that will remain valid until *numChars* characters are read.
boolean markSupported( )	Returns **true** if **mark( )/reset( )** are supported on this stream.
int read( )	Returns an integer representation of the next available character from the invoking input stream. –1 is returned when the end of the file is encountered.
int read(char *buffer*[ ])	Attempts to read up to *buffer.length* characters into *buffer* and returns the actual number of characters that were successfully read. –1 is returned when the end of the file is encountered.
abstract int read(char *buffer*[ ], int *offset*, int *numChars*)	Attempts to read up to *numChars* characters into *buffer* starting at *buffer*[*offset*], returning the number of characters successfully read. –1 is returned when the end of the file is encountered.
boolean ready( )	Returns **true** if the next input request will not wait. Otherwise, it returns **false**.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long *numChars*)	Skips over *numChars* characters of input, returning the number of characters actually skipped.

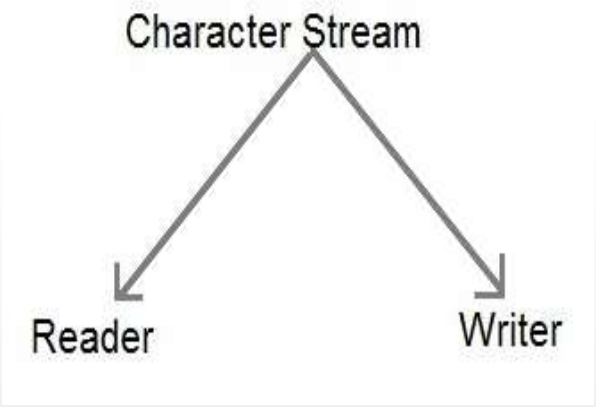
Writer

- **Writer** is an abstract class that defines streaming character output.

- All of the methods in this class throw an **IOException** in the case of errors.

Table 19-4 shows a synopsis of the methods in **Writer**.

Method	Description
Writer append(char *ch*)	Appends *ch* to the end of the invoking output stream. Returns a reference to the invoking stream.
Writer append(CharSequence *chars*)	Appends *chars* to the end of the invoking output stream. Returns a reference to the invoking stream.
Writer append(CharSequence *chars*, int *begin*, int *end*)	Appends the subrange of *chars* specified by *begin* and *end*–1 to the end of the invoking ouput stream. Returns a reference to the invoking stream.
abstract void close( )	Closes the output stream. Further write attempts will generate an **IOException**.
abstract void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(int *ch*)	Writes a single character to the invoking output stream. Note that the parameter is an **int**, which allows you to call **write** with expressions without having to cast them back to **char**.



.....

.....

FileReader

FileWriter

FileReader

- **FileReader** class creates a **Reader** that can be uses to read the contents of a file.
- Its two most commonly used constructors are:

FileReader(String *filePath*)

FileReader(File *fileObj*)

- Either can throw a **FileNotFoundException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file.

```
// Demonstrate FileReader.
   import java.io.*;
   class FileReaderDemo
   {     public static void main(String args[]) throws IOException
         {
               FileReader fr = new FileReader("FileReaderDemo.java");
                int i;
               while((i = fr.read()) != -1)
              {             System.out.println((char)i);
               }
               fr.close();
         }}
```

FileWriter

- **FileWriter** creates a **Writer** that we can use to write to a file.
- Its most commonly used constructors are:

FileWriter(String *filePath*)

FileWriter(String *filePath*, boolean *append*)

FileWriter(File *fileObj*)

FileWriter(File *fileObj*, boolean *append*)

- They can throw an **IOException**.
- If *append* is **true**, then output is appended to the end of the file.
- Creation of a **FileWriter** is not dependent on the file already existing. **FileWriter** will create the file before opening it for output when you create the object. In the case where you attempt to open a read-only file, an **IOException** will be thrown.

```
// Demonstrate FileWriter.
import java.io.*;
class FileWriterDemo
 {        public static void main(String args[]) throws IOException
         {
                  String source = "Now is the time for all good men\n"
                  + " to come to the aid of their country\n"
                 + " and pay their due taxes.";
                 char buffer[] = new char[source.length()];
                source.getChars(0, source.length(), buffer, 0);
                 FileWriter f0 = new FileWriter("file1.txt");
                  for (int i=0; i < buffer.length; i += 2) {
                                    f0.write(buffer[i]);
          }
         f0.close();
```

```
        FileWriter f1 = new FileWriter("file2.txt");
        f1.write(buffer);
        f1.close();

        FileWriter f2 = new FileWriter("file3.txt");
        f2.write(buffer,buffer.length-buffer.length/4,buffer.length/4);
         f2.close();
    }
}
```

The first, **file1.txt**, will contain every other character from the sample.
The second, **file2.txt**, will contain the entire set of characters.
Finally, the third, **file3.txt**, will contain only the last quarter

BufferedReader and BufferedWriter

The advantage of buffering is that the number of reads and writes to a physical device is reduced.

This improves the performance.

BufferedReader class extends Reader and buffers input from a character stream.

BufferedReader(Reader r)

BufferedReader(Reader r, bufSize)

The first form creates a buffered stream using a buffer with a default size.

In the second, size of the buffere is specified with bufSize.

BufferedReader and BufferedWriter

This class implements all of the methods defined by Reader.

In addition, provides the readLine() method reads newline terminated strings from a character stream.

String readLine() throws IOException

```
import java.io.*;
   class BufferedReaderDemo1
   {     public static void main(String args[]) throws IOException
         {
               FileReader fr = new FileReader("D:\\IOExamples\\MIT\\ICT\\first.txt");
               BufferedReader br = new BufferedReader(fr);
               String s;
                while((s = br.readLine()) != null)
                {
                        System.out.println(s);
               }
                fr.close();
         }
   }
```

BufferedReader and BufferedWriter

The BufferedWriter class extends Writer and buffers output to a character stream.

BufferedWriter(Writer w)

BufferedWriter(Writer w, bufSize)

The first form creates a buffered stream using a buffer with a default size.

In the second, size of the buffere is specified with bufSize.

BufferedWriter and BufferedReader

BufferedWriter class implements all of the methods defined by Writer.

In addition, provides the newline() method to output a line separator.

```
void newLine() throws IOException
```

```
import java.io.*;
class BufferedWriterDemo1
{   public static void main(String args[]) throws IOException
    {
          FileWriter fw = new FileWriter(“D:\\IOExamples\\MIT\\ICT\\first.txt");
          BufferedWriter bw = new BufferedWriter(fw);

         for(int i=0;i<12;i++)
             bw.write("Line "+i+"\n");
         bw.close();
     }
}
```

Tutorial class

- Write a java program to find the occurrence of word in a given file

Serialization

- The process of writing the state of an object to a byte stream.
- To save the state of our program to a persistent storage area, such as a file.
- At a later time, we may restore these objects by using the process of deserialization.

- Serialization is also needed to implement Remote Method Invocation (RMI).
- An object may be supplied as an argument to that remote method. The sending machine serializes the object and transmits it. The receiving machine deserializes it.

Serializable

- Only an object that implements the **Serializable** interface can be saved and restored by the serialization facilities.
- The **Serializable** interface defines no members.
  - It is simply used to indicate that a class may be serialized.
  - If a class is serializable, all of its subclasses are also serializable.
- **static** variables are not saved.

ObjectOutput

- The **ObjectOutput** interface extends the **DataOutput** interface and supports object serialization. Follwing are the methods.

Method	Description
void close( )	Closes the invoking stream. Further write attempts will generate an **IOException**.
void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(byte *buffer*[ ])	Writes an array of bytes to the invoking stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].
void write(int *b*)	Writes a single byte to the invoking stream. The byte written is the low-order byte of *b*.
void writeObject(Object *obj*)	Writes object *obj* to the invoking stream.

TABLE 19-6 The Methods Defined by **ObjectOutput**

The **writeObject( )** method is called to serialize an object.

- All of these methods will throw an **IOException** on error conditions.

ObjectOutputStream

- The **ObjectOutputStream** class extends the **OutputStream** class and implements the **ObjectOutput** interface.

- It is responsible for writing objects to a stream.

ObjectOutputStream(OutputStream *outStream*) throws IOException

- outStream is the output stream to which serialized objects will be written.

- Several commonly used methods in this class are shown in Table 19-7.
- They will throw an **IOException** on error conditions.

Method	Description
void close( )	Closes the invoking stream. Further write attempts will generate an **IOException**.
void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(byte *buffer*[ ])	Writes an array of bytes to the invoking stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].
void write(int *b*)	Writes a single **byte** to the invoking stream. The byte written is the low-order byte of *b*.
void writeBoolean(boolean *b*)	Writes a **boolean** to the invoking stream.
void writeByte(int *b*)	Writes a **byte** to the invoking stream. The byte written is the low-order byte of *b*.
void writeBytes(String *str*)	Writes the bytes representing *str* to the invoking stream.
void writeChar(int *c*)	Writes a **char** to the invoking stream.
void writeChars(String *str*)	Writes the characters in *str* to the invoking stream.
void writeDouble(double *d*)	Writes a **double** to the invoking stream.
void writeFloat(float *f*)	Writes a **float** to the invoking stream.
void writeInt(int *i*)	Writes an **int** to the invoking stream.
void writeLong(long *l*)	Writes a **long** to the invoking stream.
final void writeObject(Object *obj*)	Writes *obj* to the invoking stream.
void writeShort(int *i*)	Writes a **short** to the invoking stream.

TABLE 19-7 Commonly Used Methods Defined by **ObjectOutputStream**

ObjectInput

- The **ObjectInput** interface extends the **DataInput** interface and defines the methods shown in Table 19-8.

- It supports object serialization.

- The **readObject( )** method is called to deserialize an object.

- All of these methods will throw an **IOException** on error conditions.

- The **readObject( )** method can also throw **ClassNotFoundException**.

Method	Description
int available( )	Returns the number of bytes that are now available in the input buffer.
void close( )	Closes the invoking stream. Further read attempts will generate an **IOException**.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ])	Attempts to read up to *buffer.length* bytes into *buffer*, returning the number of bytes that were successfully read. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes that were successfully read. –1 is returned when the end of the file is encountered.
Object readObject( )	Reads an object from the invoking stream.
long skip(long *numBytes*)	Ignores (that is, skips) *numBytes* bytes in the invoking stream, returning the number of bytes actually ignored.

TABLE 19-8 The Methods Defined by **ObjectInput**

ObjectInputStream

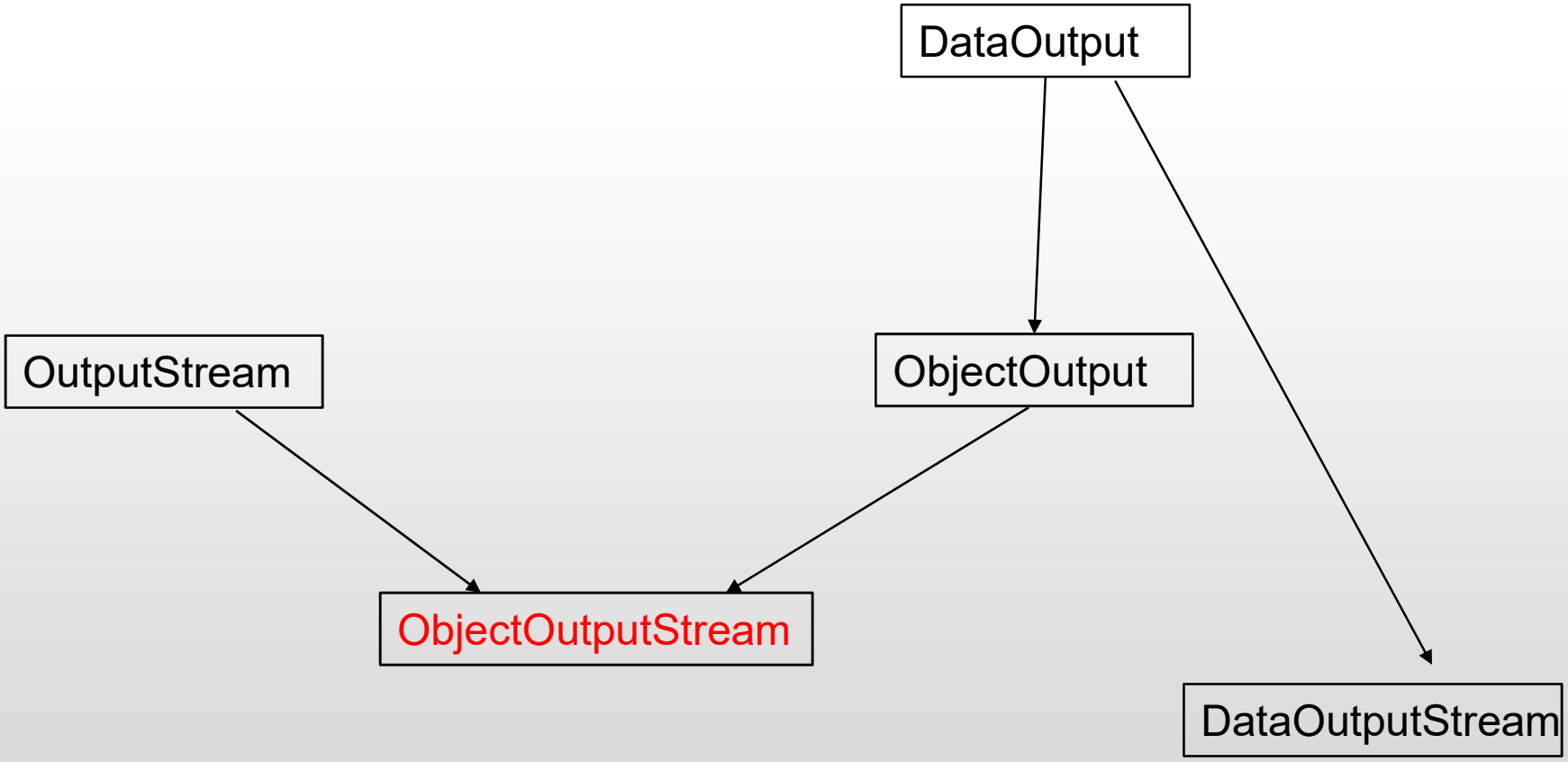
- The **ObjectInputStream** class extends the **InputStream** class and implements the **ObjectInput** interface.
- **ObjectInputStream** is responsible for reading objects from a stream.

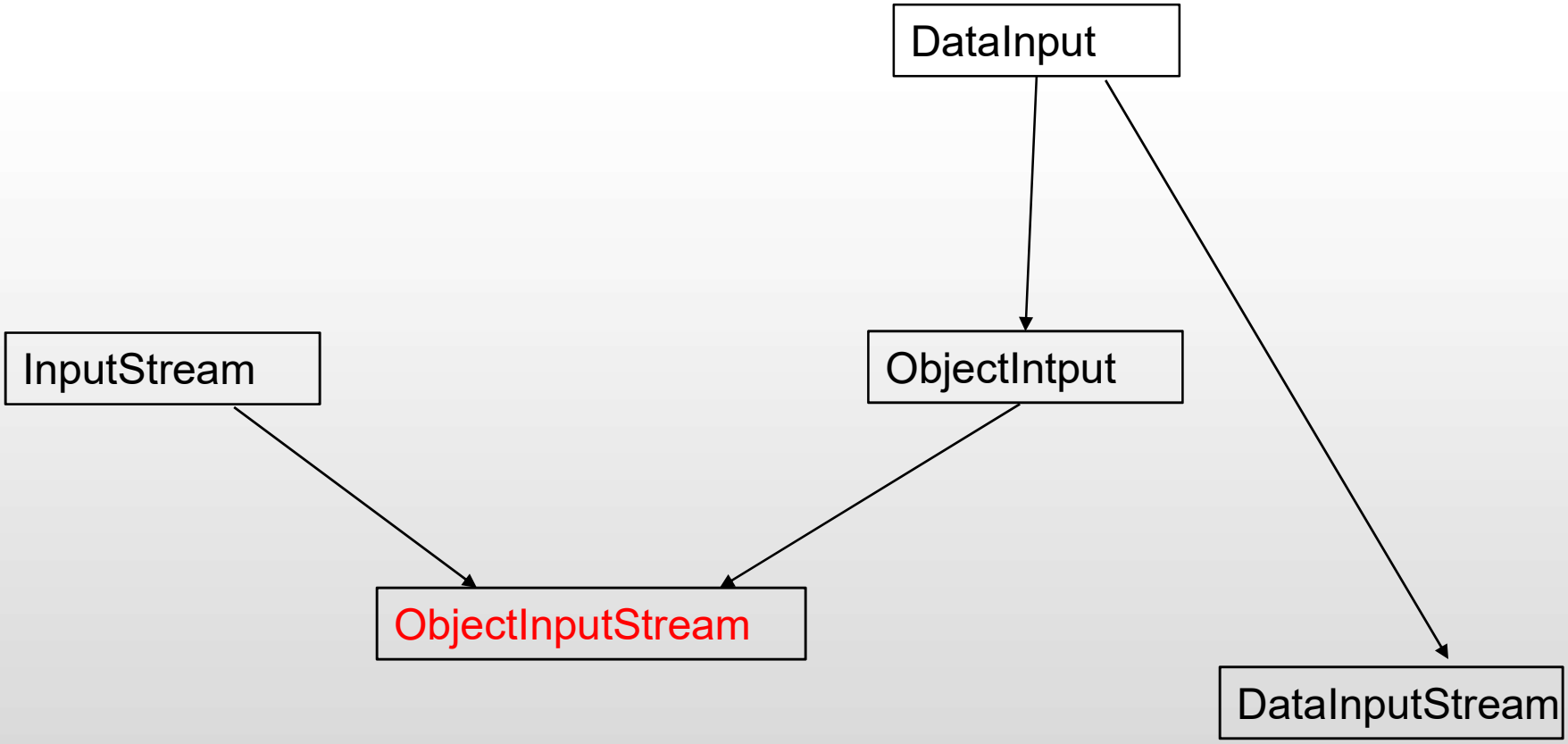
ObjectInputStream(InputStream *inStream*) throws IOException

- *inStream* is the input stream from which serialized objects should be read.
- Several commonly used methods in this class are shown in Table 19-9.
- They will throw an **IOException** on error conditions. The **readObject( )** method can also throw **ClassNotFoundException**.

Method	Description
int available( )	Returns the number of bytes that are now available in the input buffer.
void close( )	Closes the invoking stream. Further read attempts will generate an **IOException**.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered.
boolean readBoolean( )	Reads and returns a **boolean** from the invoking stream.
byte readByte( )	Reads and returns a **byte** from the invoking stream.
char readChar( )	Reads and returns a **char** from the invoking stream.
double readDouble( )	Reads and returns a **double** from the invoking stream.
float readFloat( )	Reads and returns a **float** from the invoking stream.
void readFully(byte *buffer*[ ])	Reads *buffer.length* bytes into *buffer*. Returns only when all bytes have been read.
void readFully(byte *buffer*[ ], int *offset*, int *numBytes*)	Reads *numBytes* bytes into *buffer* starting at *buffer*[*offset*]. Returns only when *numBytes* have been read.
int readInt( )	Reads and returns an **int** from the invoking stream.
long readLong( )	Reads and returns a **long** from the invoking stream.
final Object readObject( )	Reads and returns an object from the invoking stream.
short readShort( )	Reads and returns a **short** from the invoking stream.
int readUnsignedByte( )	Reads and returns an unsigned **byte** from the invoking stream.
int readUnsignedShort( )	Reads and returns an unsigned **short** from the invoking stream.

TABLE 19-9 Commonly Used Methods Defined by **ObjectInputStream**


DataOutput
OutputStream
ObjectOutput
ObjectOutputStream
DataOutputStream


DataInput
InputStream
ObjectIntput
ObjectInputStream
DataInputStream

```
class MyClass implements Serializable
{
        String s;
        int i;
        double d;
        public MyClass(String s, int i, double d)
        {       this.s = s;
                this.i = i;
                this.d = d;
        }
        public String toString()
        {       return "s=" + s + "; i=" + i + "; d=" + d;
        }
}
```

```
public class SerializationDemo
{    public static void main(String args[])
     {    // Object serialization
          try
          {
               MyClass object1 = new MyClass("Hello", -7, 2.7e10);
               System.out.println("object1: " + object1);
               FileOutputStream fos = new FileOutputStream("serial");
               ObjectOutputStream oos = new ObjectOutputStream(fos);
               oos.writeObject(object1);
               oos.flush();
               oos.close();
          }
          catch(IOException e)
          {  System.out.println("Exception during serialization: " + e);
                                   System.exit(0);  }
```

```
        // Object deserialization
        try
        {
                MyClass object2;
                FileInputStream fis = new FileInputStream("serial");
                ObjectInputStream ois = new ObjectInputStream(fis);
                object2 = (MyClass)ois.readObject();
                ois.close();
                System.out.println("object2: " + object2);
        }
        catch(Exception e) {  System.out.println("Exception during deserialization: " + e);
                                System.exit(0); }
    }
}
```

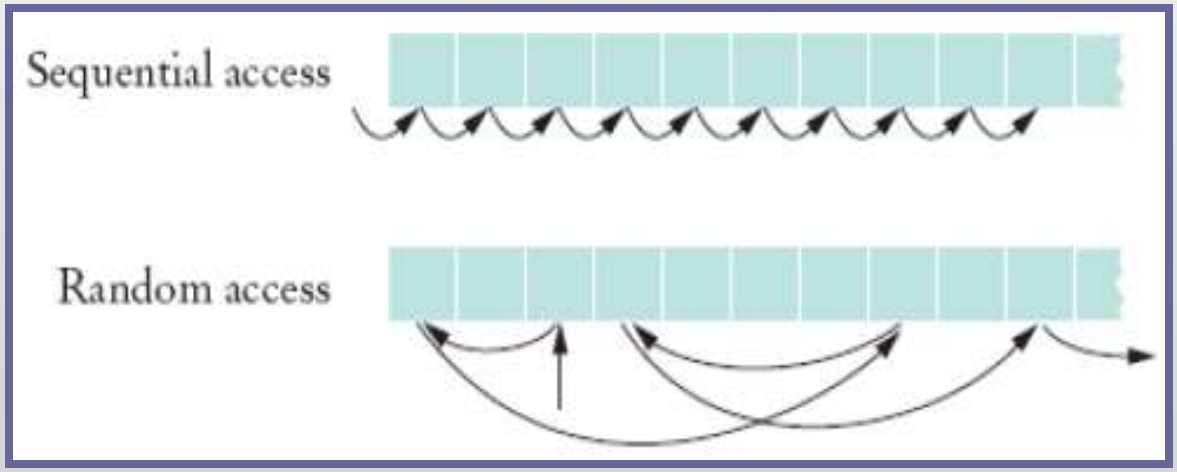
Create a class student with name and sessional mark as its data members, use constructor to initialize the object and a method to display the student details.

Write a program to write 5 student objects into a “student.txt” file.

Also read the contents of the file and display only those students detail whose sessional marks are above 10.

RandomAccessFile

Random access files are files in which records can be accessed in any order



RandomAccessFile

RandomAccessFile class to allow a file to be read from and write to at random locations.

RandomAccessFile is special because it supports positioning requests: we can position the *file pointer* within the file.

It is not derived from **InputStream** or **OutputStream**.

It implements the interfaces **DataInput** and **DataOutput**, which define the basic I/O methods.

RandomAccessFile(File *fileObj*, String *access*) throws FileNotFoundException

RandomAccessFile(String *filename*, String *access*) throws FileNotFoundException

access determines what type of file access is permitted.

If it is "r", then the file can be read, but not written.

If it is "rw", then the file is opened in read-write mode.

seek( ) : to set the current position of the file pointer within the file:

void seek(long *newPos*) throws IOException

long getFilePointer() throws IOException

int skipBytes(int n)

void setLength(long *len*) throws IOException

long length()

returns the length of the file in bytes.

```
int read();
int read(byte buffer[]);
int read(byte buffer[], int  index, int size);
```

```
void write(int b);
void write(byte buffer[]);
void write(byte buffer[], int  index, int size);
```