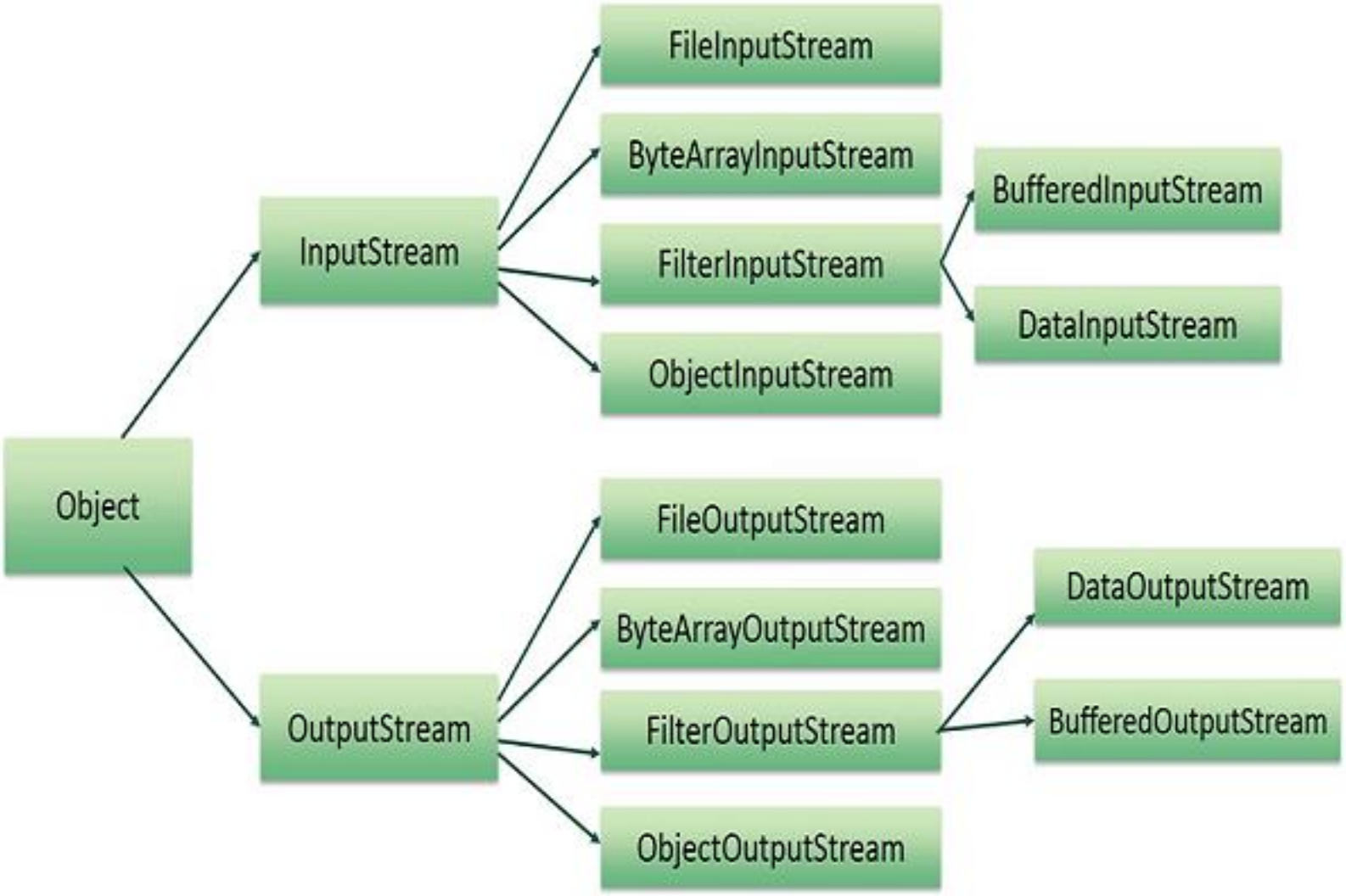
Java
INPUT/OUTPUT


Object
InputStream
FileInputStream
ByteArrayInputStream
FilterInputStream
ObjectInputStream
BufferedInputStream
DataInputStream
OutputStream
FileOutputStream
ByteArrayOutputStream
FilterOutputStream
ObjectOutputStream
DataOutputStream
BufferedOutputStream

File Input – Output

Creating File instance

- To create the `File` class instance, use:
  - **`File(String pathname)`**
  - **`File(String dir, String subpath)`**
  - **`File(File dir, String subpath)`**
- To separate directories in UNIX, use ‘/’,
  - Windows - ‘\’

File Input – Output

Main File class methods

`boolean exists()`	Returns true, if file or directory exists, otherwise returns `false`
`String getAbsolutePath()`	Returns absolute pathname to file or directory. At the same time if in the file name in the constructor a relative pathname is specified, the corresponding part of the path is stored in the returned string
`String getCanonicalPath()`	Returns absolute pathname to file or directory. At the same time if in the file name in the constructor a relative pathname is specified, the corresponding part of the path is replaced with a canonical pathname in the returned string, without elements of relative pathname. Throws IOException if the canonical pathname cannot be constructed
`String getName()`	Returns the name of the file or directory, where the name is the last name in the pathname's name sequence

File Input – Output

Main File class methods

`String getParent()`	Returns the name of the directory, where the `File` is stored
`boolean isDirectory()`	Returns `true`, if `File` denotes the directory that exists in the file system
`boolean isFile()`	Returns true, if File denotes the file that exists in the file system
`String[] list()`	Returns array of strings naming the files and directories denoted by this `File` instance, where `File` may denote not only file, but the directory as well

File Input – Output

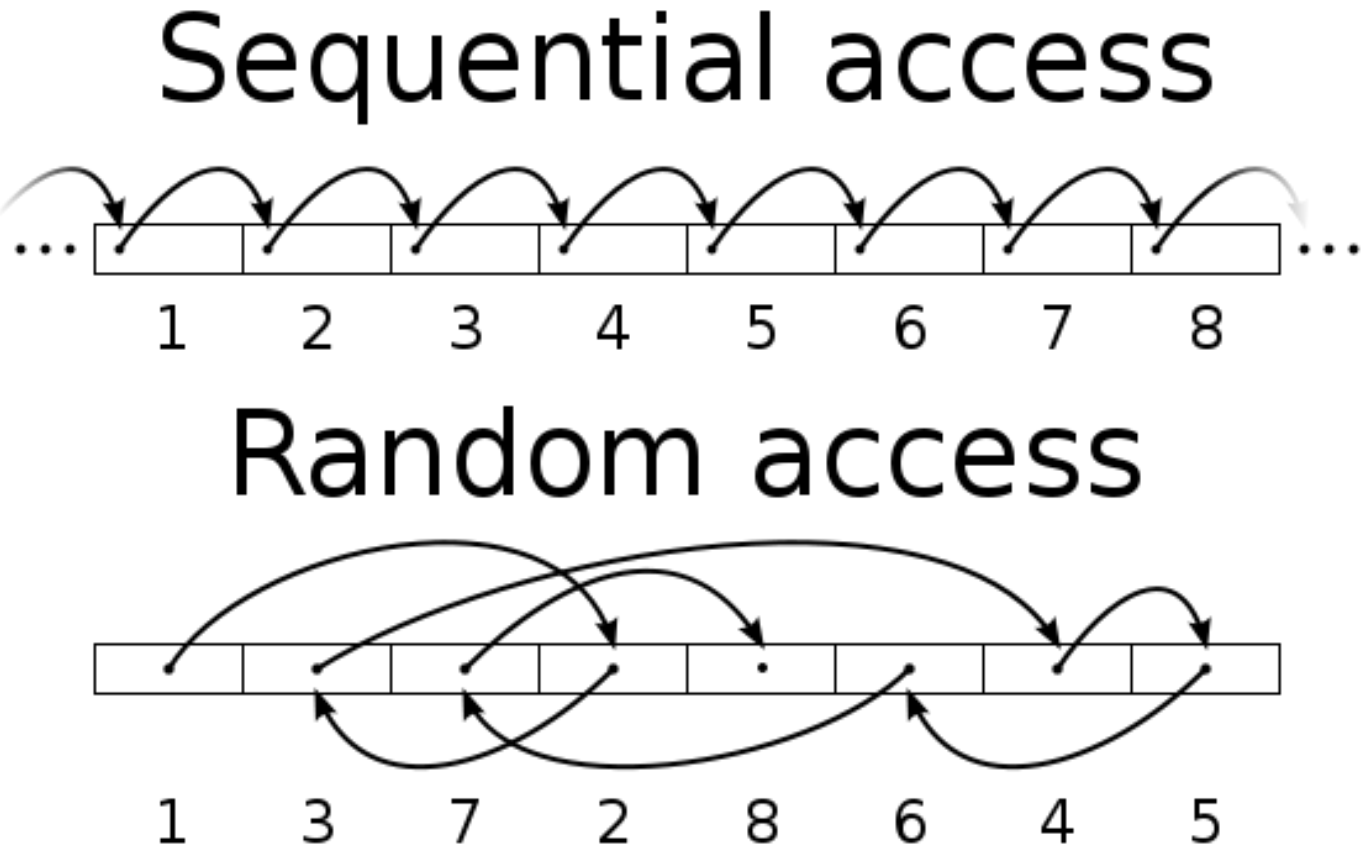
Main File class methods

`boolean canRead()`	Returns `true`, if the file and directory can be read by the application
`boolean canWrite()`	Returns `true`, if the file or directory can be modified by the application
`boolean createNewFile()`	Creates a new, empty file, if a file with this name does not yet exist. Returns `true`, if file was successfully created
`boolean delete()`	Deletes the file or directory, returns `false`, if the file wasn't deleted
`long length()`	Returns the length of the file
`boolean mkdir()`	Creates the directory with a pathname specified by current `File` instance
`boolean renameTo(File`*`newname`*`)`	Returns the file or directory. Returns `true`, if renaming succeeded; otherwise `false`

RANDOM ACCESS

File Input – Output

RandomAccessFile class



File Input – Output

RandomAccessFile constructors

`RandomAccessFile(String file, String mode)`

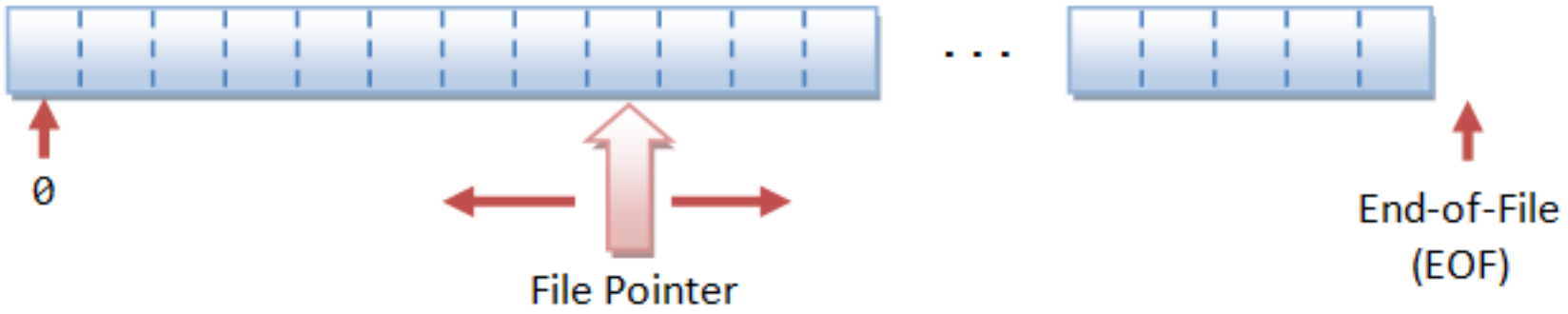
`RandomAccessFile(File file, String mode)`

The `mode` argument takes “r” value to open file for reading only; “rw” – for reading and writing. There are other rws and rwd modes for immediate data writing to disk after each I/O operation.

File Input – Output

Working with RandomAccessFile

- Once the instance is created, the file pointer can be set to a specified position with respect to the beginning; current position and file length can be received.



File Input – Output

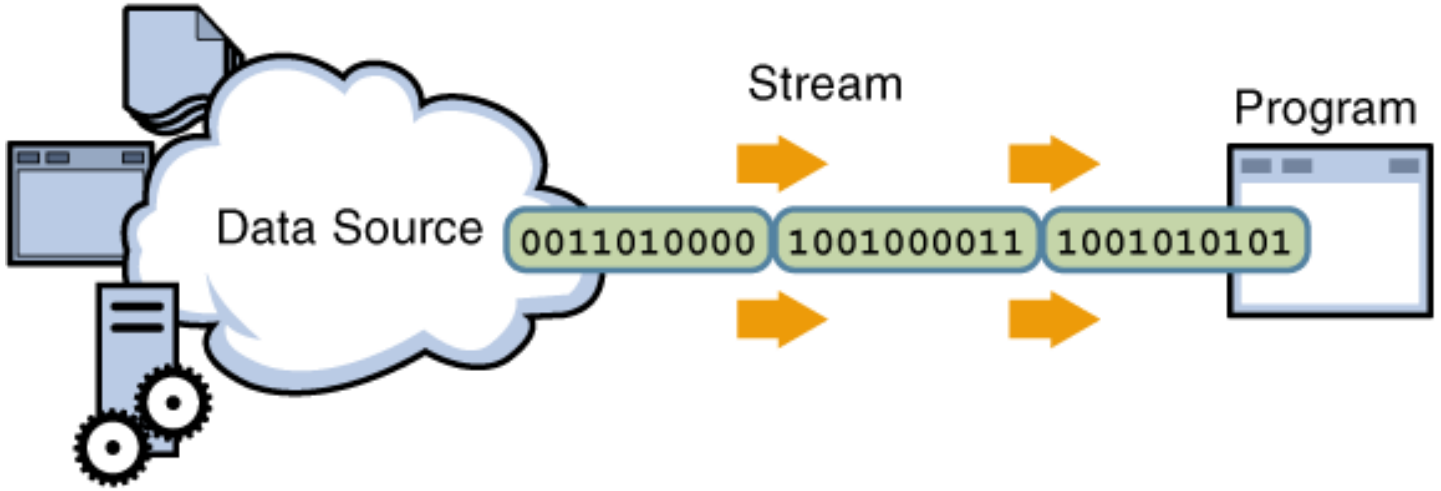
Working with RandomAccessFile

`long getFilePointer() throws IOException`	Returns the current offset in this file in bytes. The next read and write occur at this offset
`long length() throws IOException`	Returns the length of this file, measured in bytes
`void seek(long` ***`position`***`) throws IOException`	Sets the file-pointer offset, measured in bytes. The next read and write occur at this offset. The file begins from **0** offset

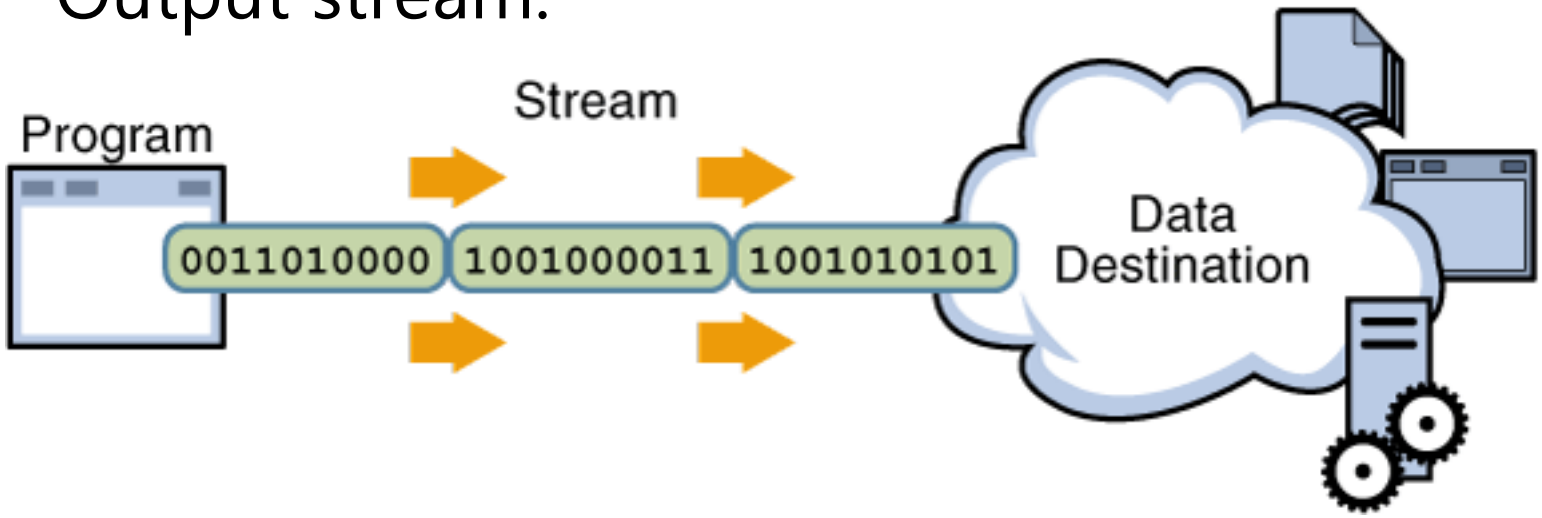
I/O STREAMS

Streams

Input stream:

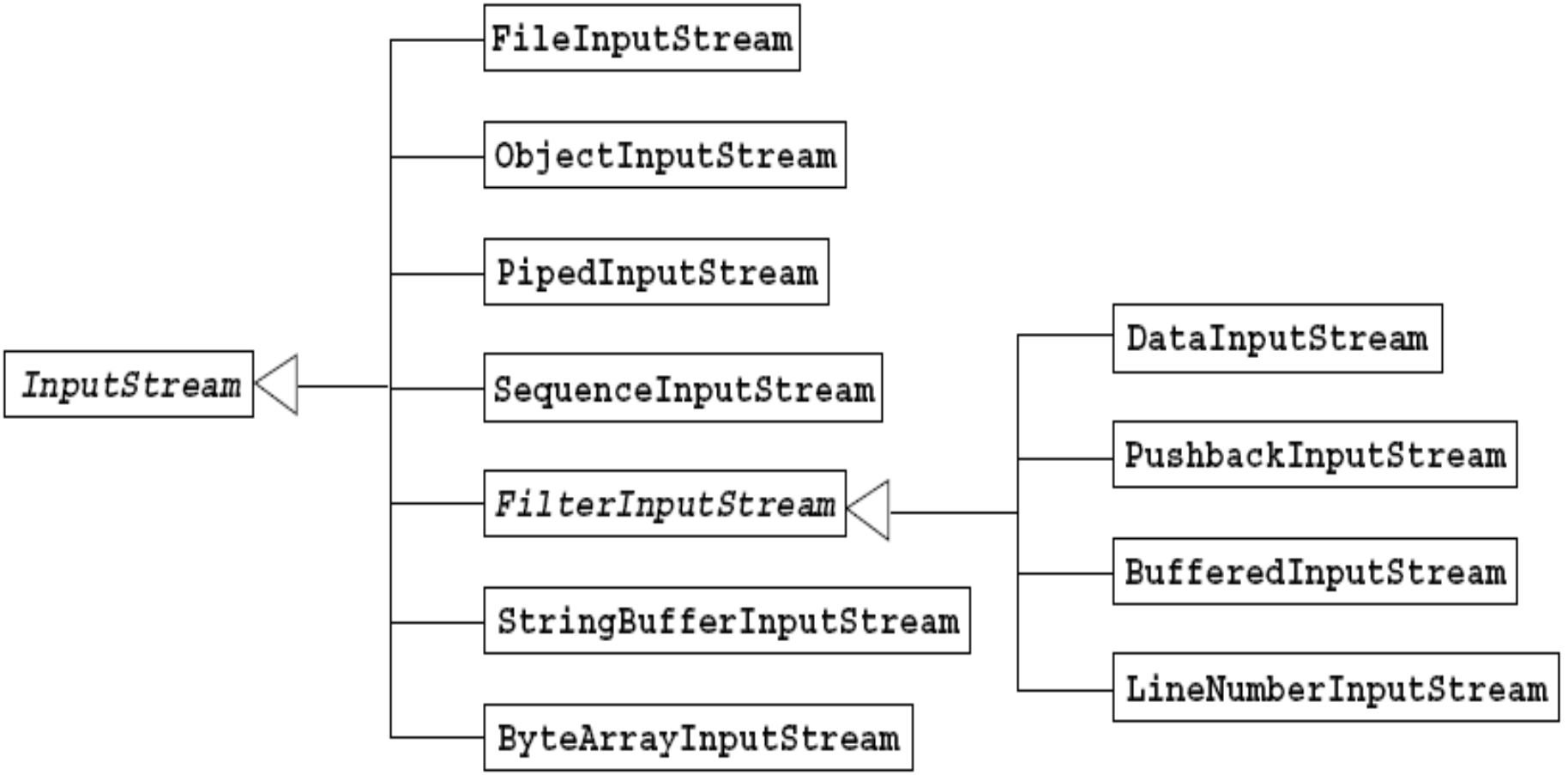


Output stream:



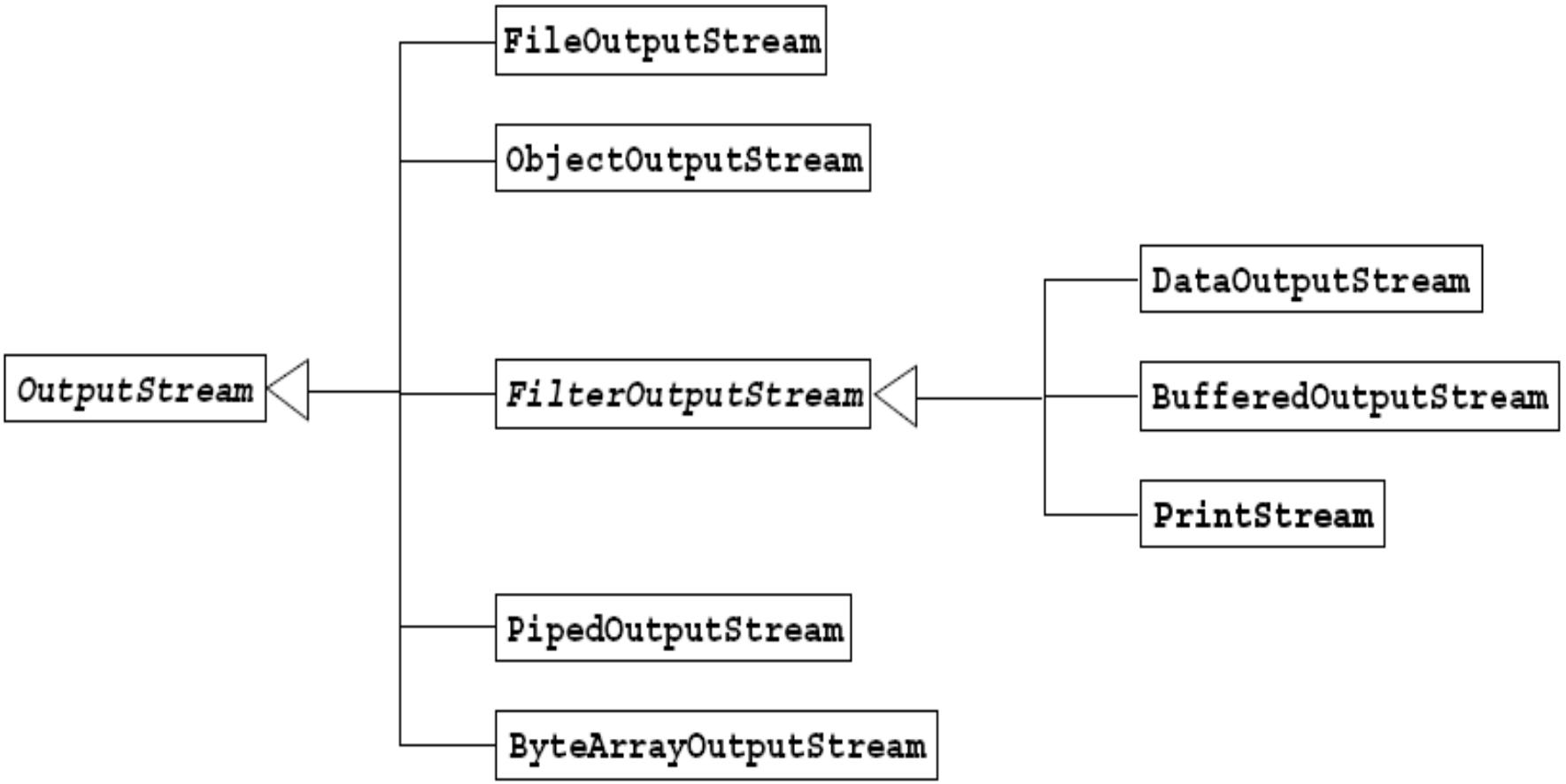
Streams

InputStream hierarchy



Streams

OutputStream hierarchy



Streams

Low-level streams

- Examples:
  - **`FileInputStream`**, **`FileOutputStream`**
  - **`ByteArrayInputStream`**, **`ByteArrayOutputStream`**
  - Streams reading and writing from TCP socket

Streams

Low-level streams

- All low-level streams widen the `InputStream` & `OutputStream` classes that provide following methods:

`void write(intb) throws IOException`	Writes low byte b.
`void write(byte bytes[]) throws IOException`	Writes all bytes from the array *`bytes`*`[]` of the *`bytes`*`[]` type
`void write(byte bytes[], int offset, int len) throws IOException`	Writes *`len`* byte from the array *`bytes`*`[]`, beginning from *`offset`*

Streams

FileInputStream

- Reading files using stream model is performed through the `FileInputStream` class
- Constructors:
  - **`FileInputStream(String pathname)`**
  - **`FileInputStream(File file)`**
- Once the instance is created one or several bytes can be read:

Streams

FileInputStream example

```
byte b;
byte bytes[] = new byte[100];
byte morebytes[] = new byte[50];
try {
   FileInputStream fis =
         new FileInputStream("fname");
   b = (byte) fis.read(); // Single byte
   fis.read(bytes); // Fill the array
   // read 1st 20 elements
   fis.read(morebytes, 0, 20);
   fis.close();
} catch (IOException e) {
   e.printStackTrace();
}
```

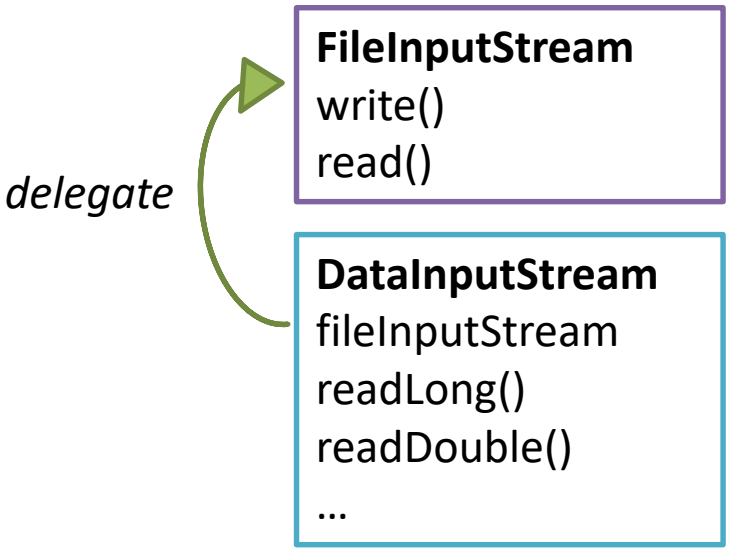
Streams

ByteArrayInputStream & ByteArrayOutputStream

- **`ByteArrayInputStream`** and **`ByteArrayOutputStream`** allow reading and writing byte arrays
- This is done without any I/O system operations. These classes are useful when dealing with sequences of bytes

Streams

High-level streams: decorator pattern

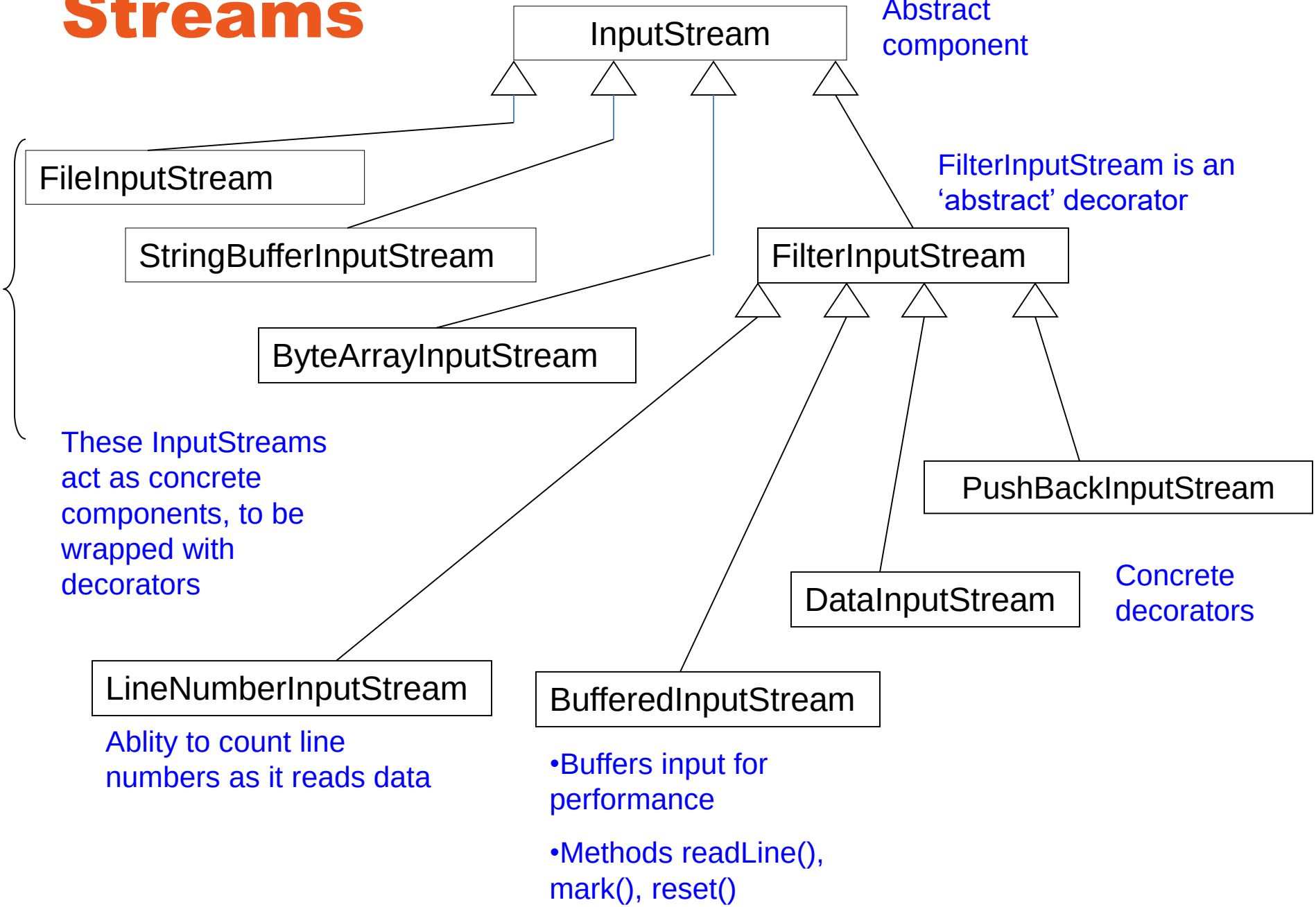


- FilterXXXStream does not access any specific data source. Other streams, such as low-level streams, are used as constructor's arguments
- During their work they read certain amount of bytes from a source stream and perform certain conversion operations

```
FileInputStream fis = new FileInputStream("fname");
DataInputStream dis = new DataInputStream(fis);

// Read
double d = dis.readDouble();
```

Streams


InputStream
Abstract component
FileInputStream
StringBufferInputStream
ByteArrayInputStream
These InputStreams act as concrete components, to be wrapped with decorators
FilterInputStream
FilterInputStream is an 'abstract' decorator
PushBackInputStream
DataInputStream
Concrete decorators
LineNumberInputStream
Ablity to count line numbers as it reads data
BufferedInputStream
•Buffers input for performance
•Methods readLine(), mark(), reset()

Streams

DataInputStream

- The DataInputStream class provides following methods:

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

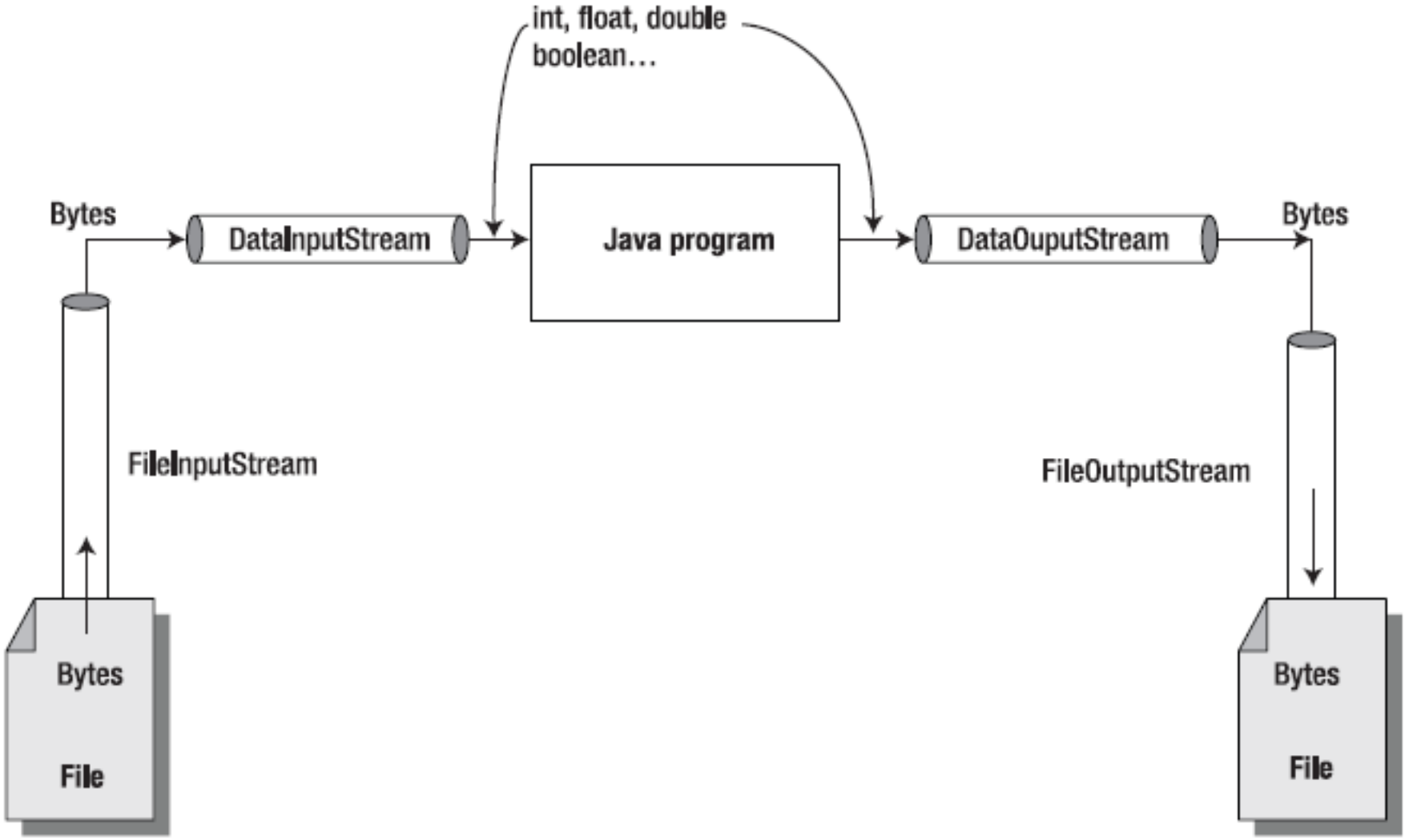
Streams

DataInputStream. Example

```
try {
    // Construct the chain
    FileInputStream fis = new FileInputStream("fname");
    DataInputStream dis = new DataInputStream(fis);
    // Read
    double d = dis.readDouble();
    int i = dis.readInt();
    String s = dis.readUTF();
    // Close the chain
    dis.close(); // Close dis first, because it
    fis.close(); // was created last
} catch (IOException e) {
}
```

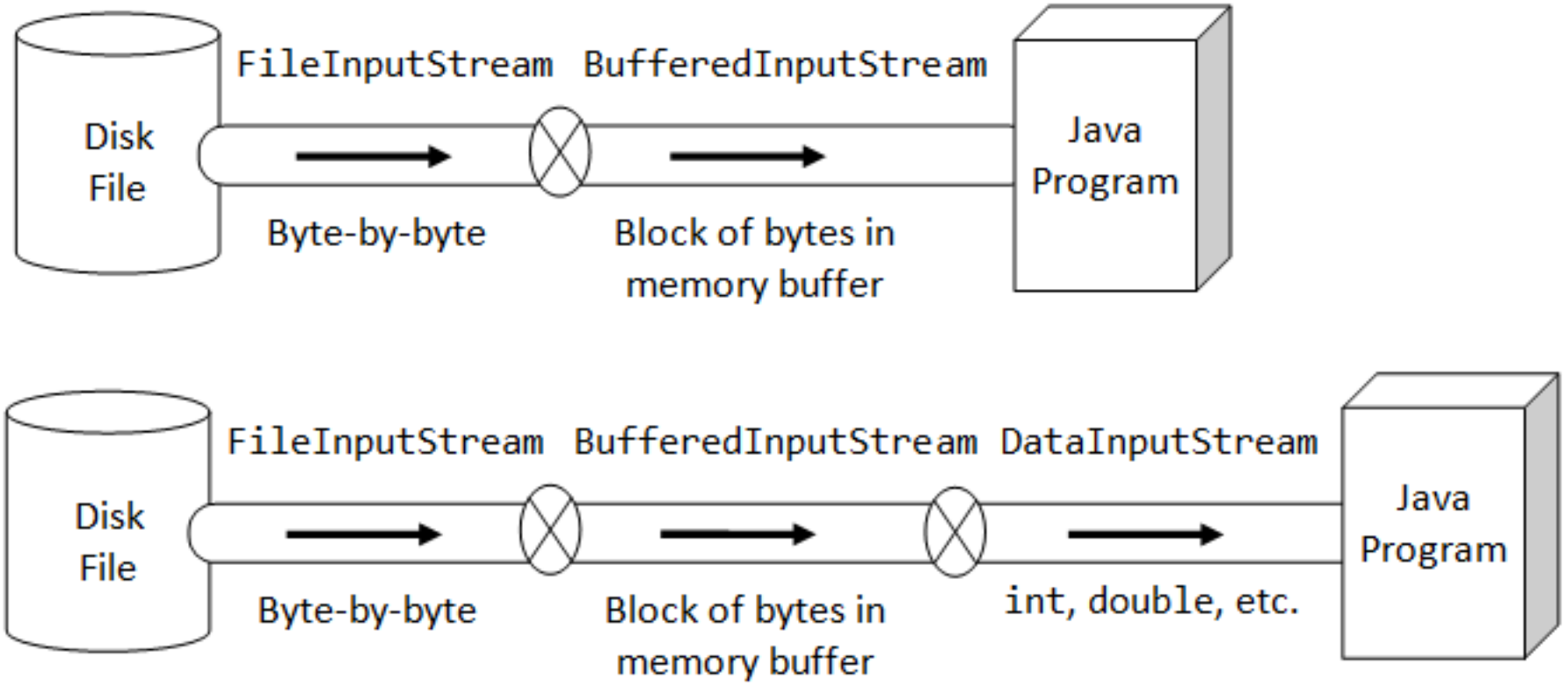
Streams

DataInputStream



Streams

BufferedInputStream

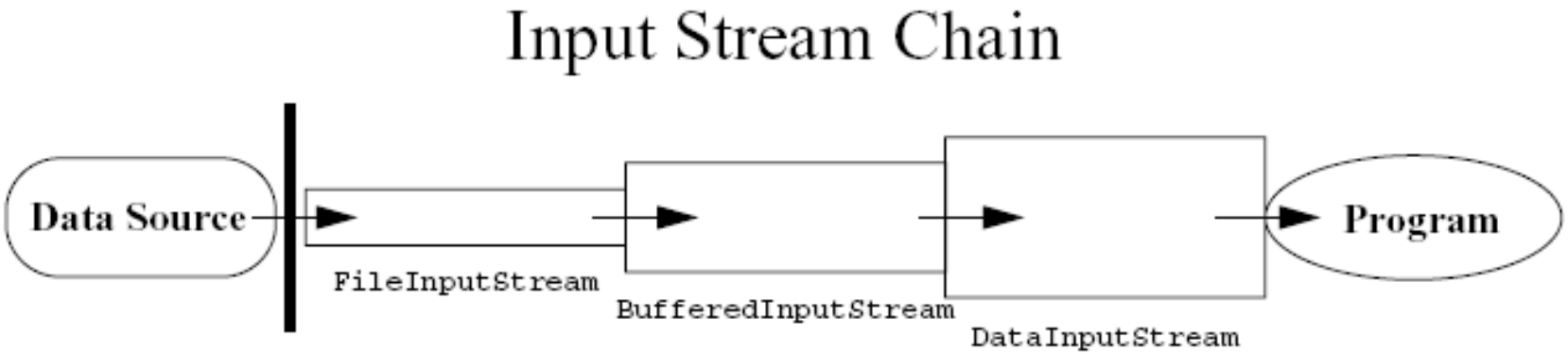


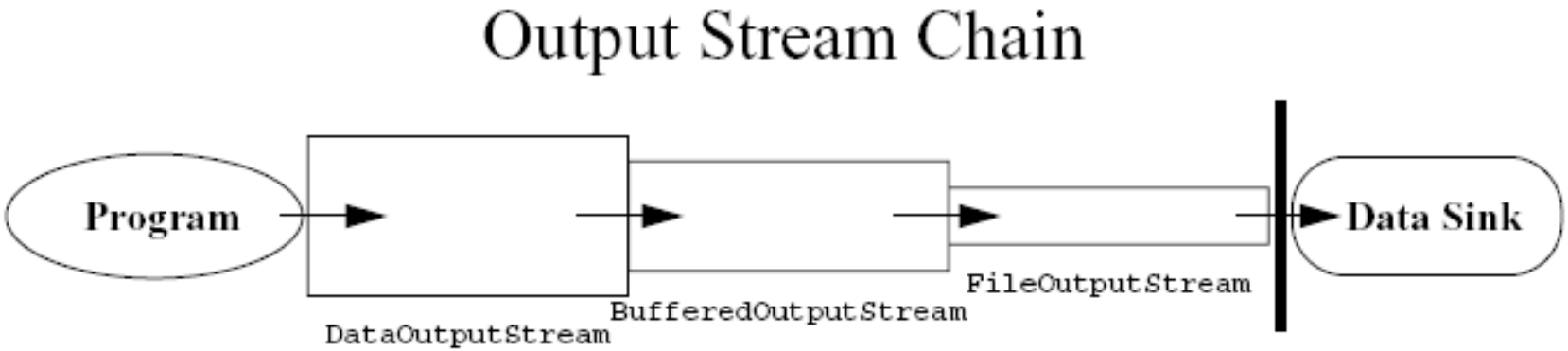
STREAM CHAINING

Streams

Stream chaining

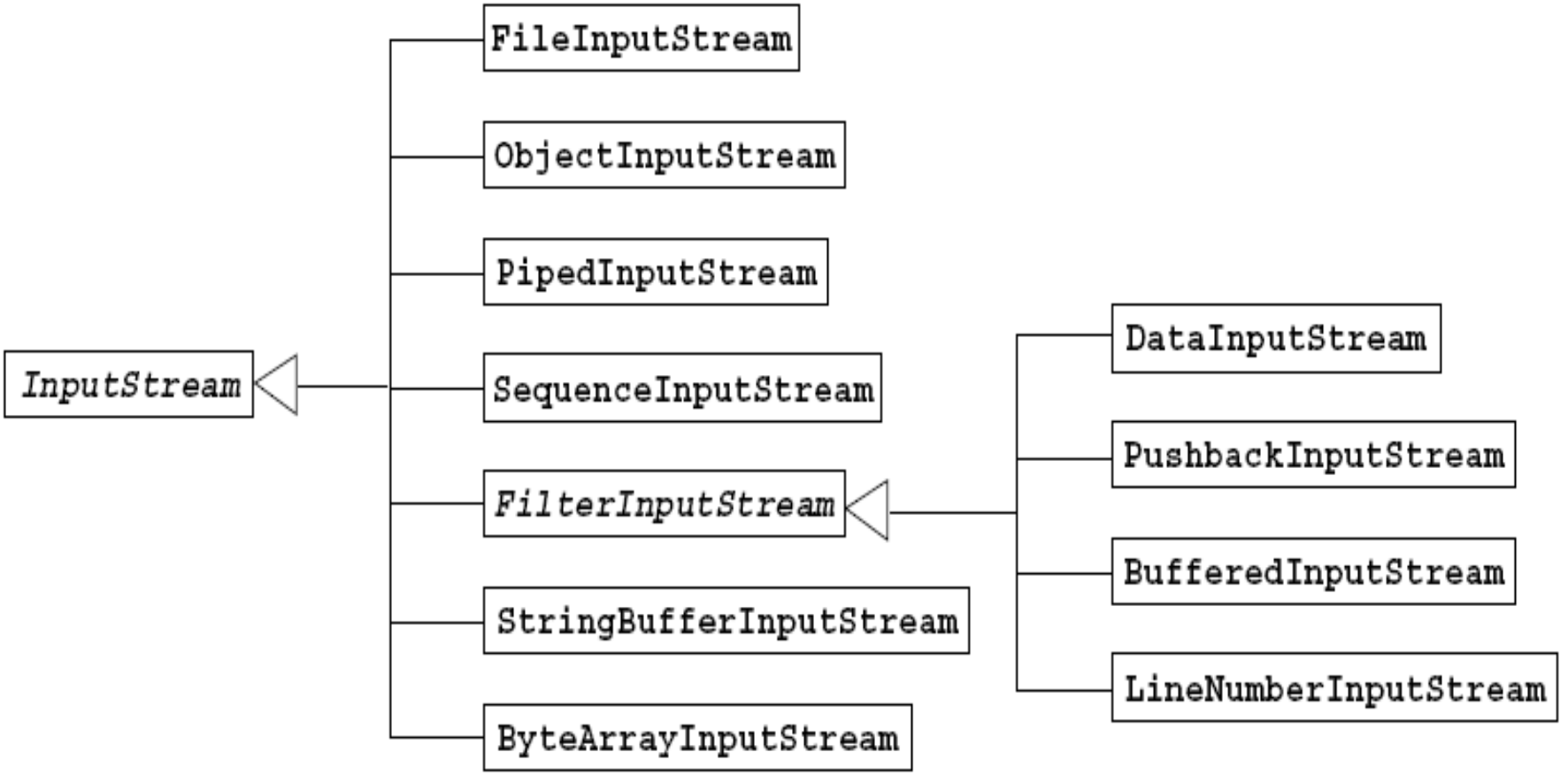
Connection of low-level and high-level streams is called stream chaining:





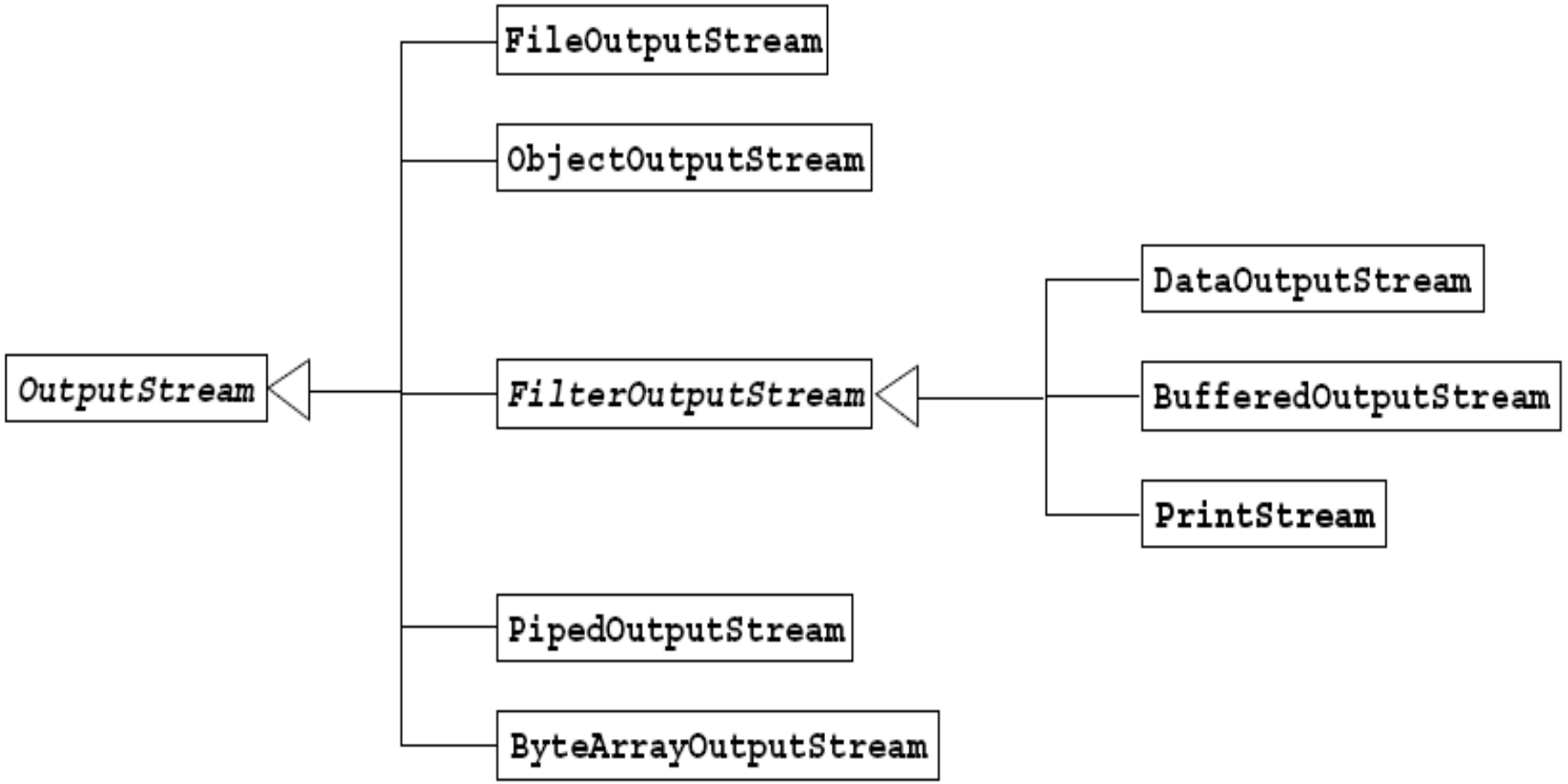
Streams

InputStream hierarchy



Streams

OutputStream hierarchy



HANDLING EXCEPTIONS IN STREAMS

Work with the resources: finally

```
InputStream input = null;

try {

  input = new FileInputStream(”file.txt");

  // do something with the stream

} catch(IOException e){ // first catch block
  throw new WrapperException(e);
} finally {
  try {
     if(input != null) input.close();
  } catch(IOException e) {  // second catch block
     throw new WrapperException(e);
  }
}
```

Work with resources: try-with-resources

```
private static void printFileJava7() throws IOException {

    try(FileInputStream input = new FileInputStream("file.txt")){

        int data = input.read();
        while(data != -1){
            System.out.print((char) data);
            data = input.read();
        }
    }
}
```

Work with resources: try-with-resources multiple

```
private static void printFileJava7() throws IOException {

    try(  FileInputStream input = new FileInputStream("file.txt");
          BufferedInputStream bufferedInput = new BufferedInputStream(input)
    ) {

        int data = bufferedInput.read();
        while(data != -1){
              System.out.print((char) data);
              data = bufferedInput.read();
        }
    }
}
```

Using try-with-resources is available for any resource implementing java.lang.AutoCloseable

Work with resources: AutoClosable

```
public interface AutoClosable {
    public void close() throws Exception;
}

public class MyAutoClosable implements AutoCloseable {
    public void doIt() {
        System.out.println("MyAutoClosable doing it!");
    }

    @Override
    public void close() throws Exception {
        System.out.println("MyAutoClosable closed!");
    }
}

private static void myAutoClosable() throws Exception {
    try(MyAutoClosable myAutoClosable = new MyAutoClosable()){
        myAutoClosable.doIt();
    }
}
```

```
MyAutoClosable doing it!
MyAutoClosable closed!
```

READERS AND WRITERS STREAMS

Readers & Writers

Encodings

- Encoding is the process of putting characters into certain number of bytes
- There are one-byte encodings (for instance, Windows 1251, KOI-8R). Only one byte is needed to store one character. There are total 256 different character values

Readers & Writers

Encodings

- Unicode encoding uses two bytes for one character. This means that it can store 65536 possible characters (Russian, Chinese, Greek)
- This encoding is used for one character in the `String` class

Readers & Writers

Readers & Writers

- Reader and Writer implementations:

`CharArrayReader` and `CharArrayWriter`	Reads and writes `char` arrays
`PipedReader` and `PipedWriter`	Provide mechanism of streams interaction
`StringReader` and `StringWriter`	Reads and writes the strings

Readers & Writers

Reader

- Abstract classes `Reader` and `Writer` contain following methods to read and write the characters:

`int read() throws IOException`	Returns the next character (stored in 16 bits of `int` type) or -1 if the end of the stream has been reached
`int read(char dest[]) throws IOException`	Reads as many characters as needed to fill the *`dest`*[] array. Returns the number of characters read or -1 if the stream has been reached
`int read(char dest[], int offset, int len) throws IOException`	Reads *`len`* characters in the *`dest`*`[]` array from the *`offset`*. Returns the number of characters read or -1 if the stream has been reached

Readers & Writers

Writer

`void write(int ch) throws IOException`	Writes a single character that is specified as *`ch`* in 16 low-order bits
`void write(String str) throws IOException`	Writes the string *`str`*
`void write(String str, int offset, int len) throws IOException`	Writes substring *`str`* with lenght *`len`*, which starts at the *`offset`* position
`void write(char chars[]) throws IOException`	Writes characters of the *`chars[]`* array
`void write(char chars[], int offset, int len) throws IOException`	Writes *`len`* characters from the array *`chars[]`*, starting at *`offset`*

Readers & Writers

FileReader and FileWriter

- Classes `FileReader` and `FileWriter` make it possible to read the contents of a file as a stream of characters in default system encoding
- Constructors:
  - **`FileReader(String pathname)`**
  - **`FileReader(File file)`**
  - **`FileWriter(String pathname)`**
  - **`FileWriter(File file)`**

Readers & Writers

Low-level readers and writers

Other low-level readers and writers:

- `CharArrayReader` and `CharArrayWriter` read and write character arrays
- `PipedReader` and `PipedWriter` allow streams to communicate
- `StringReader` and `StringWriter` read and write strings

Readers & Writers

High-level readers and writers

- High-level readers and writers inherit from `Reader` and `Writer` classes. However, their constructors take other low-level objects as parameters and form chains
- `BufferedReader` and `BufferedWriter` provides buffering that speed up I/O operations, because large blocks of character are read and written at a time
- Classes `InputStreamReader` and `OutputStreamWriter` turn byte streams into sequences of Unicode characters. By default, these classes use system encoding. Alternate constructors are used to specify other encodings.

Readers & Writers

High-level readers and writers

- The `LineNumberReader` class deals with input sequence of characters as with a set of text lines. The `readLine()` method returns the next read line of characters. The class keeps track of line numbers of the read characters

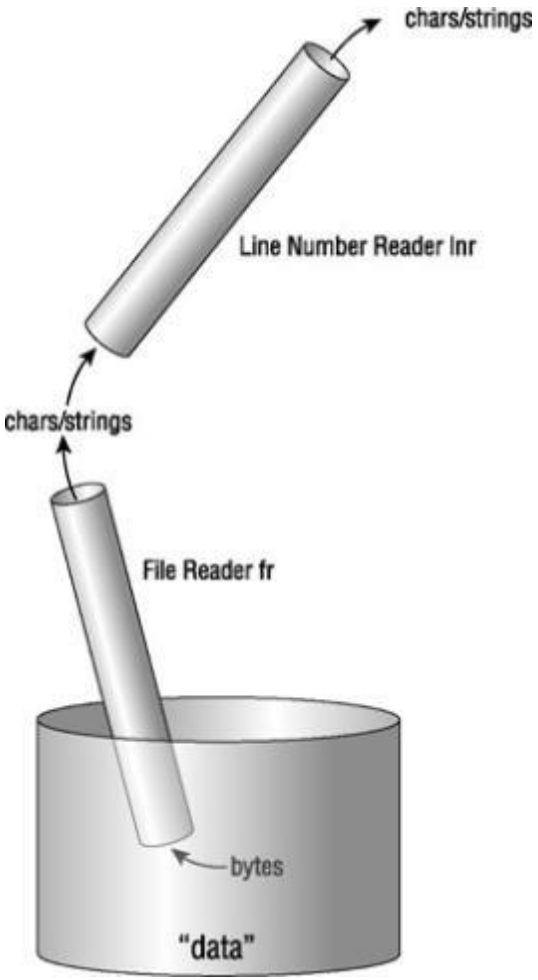
Readers & Writers

High-level readers and writers

```
try {
    FileReader fr = new FileReader("data");
    LineNumberReader lnr =
              new LineNumberReader(fr);
    String s;
    while ((s = lnr.readLine()) != null) {
       System.out.println(lnr.getLineNumber()
          +" : " + s);
    }
    lnr.close();
    fr.close();
}
catch (IOException e) {
    e.printStackTrace();
}
```

Readers & Writers

High-level readers and writers



Streams, Readers & Writers

Summary

Type	Character Streams	Byte Streams
Buffering	`BufferedReader` `BufferedWriter`	`BufferedInputStream` `BufferedOutputStream`
Filtering	`FilterReader` `FilterWriter`	`FilterInputStream` `FilterOutputStream`
Converting between bytes and character	`InputStreamReader` `OutputStreamWriter`	
Performing object serialization		`ObjectInputStream` `ObjectOutputStream`
Performing data conversion		`DataInputStream` `DataOutputStream`
Counting	`LineNumberReader`	`LineNumberInputStream`
Peeking ahead	`PushbackReader`	`PushbackInputStream`
Printing	`PrintWriter`	`PrintStream`

SYSTEM CLASS OBJECTS: OUT, IN, ERR

PRINTSTREAM CLASS

PrintStream

The System class objects: out, in, err

- The **`System.out`** variable allows printing strings to standard output (console). **`System.out`** is the **`PrintStream`** class object
- The **`System.err`** variable is used to display information in standard “error” output stream. **`System.err`** is the **`PrintStream`** class object
- The **`System.in`** variable is used to read from the standard input stream. **`System.in`** is the **`PrintStream`** class object
- JVM initiates these objects during startup

PrintStream

The PrintStream class

- The **`PrintStream`** class inherits from **`FilterOutputStream`** and provides handy methods to print data of various types in default system encoding
- The **`PrintStream`** class has a number of the overridden **`print`**`()` methods printing information of this type and the **`println()`** method that terminates the line
- The **`print`**`()` methods return **`String.valueOf(type)`**.
- The **`print(Object o)`** method returns **`o.toString()`**

The PrintStream class. Example

```
public static void main(String[] args) {
    int var = 5;
    Date now = new Date();
    System.out.println(var); // 5
    System.out.println(now);
    // Mon Feb 01 17:30:23 NOVT 2010

    // This equals to:
    PrintStream ps = System.out;
    ps.println(var);
    ps.println(now);

}
```

PrintStream

Example of reading from System.in

```
public static void main(String[] args) {
    String s;
InputStreamReader ir=new InputStreamReader(System.in);
    BufferedReader in = new BufferedReader(ir);

    try {
        s = in.readLine();
        while (s != null) {
            System.out.println("Read: " + s);
            s = in.readLine();
        }
        in.close();
    } catch (IOException e) { // Catch any IO exceptions.
        e.printStackTrace();
    }
}
```