Files & Streams

Vũ Thị Hồng Nhạn

(vthnhan@vnu.edu.vn)

Dept. of Software Engineering,

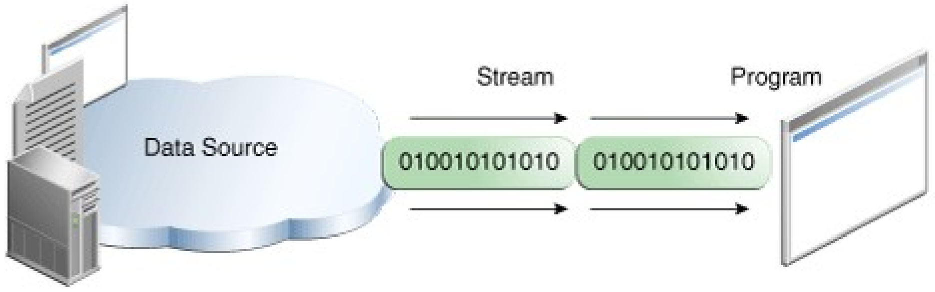
Faculty of Information Technology, UET

Vietnam National Univ., Hanoi

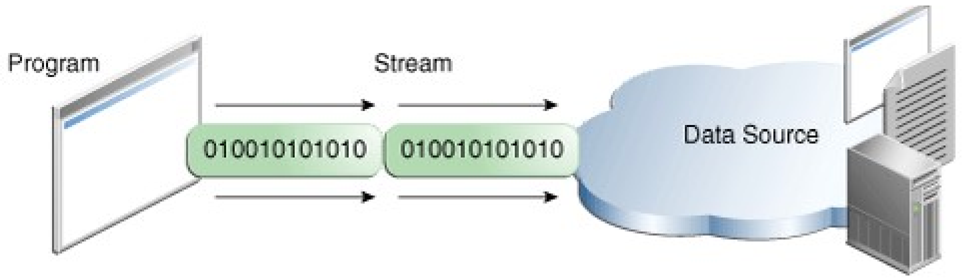
Contents

I/O streams

- **A stream** is a sequence of data
- **Streams** support many different kinds of data, including *bytes, primitive data types, characters*, and *objects*
- **An I/O stream** represents many different kinds of sources and destinations (e,g. disk files, devices, other programs)
  - A program uses an Input stream to read data **from** a source, one item at a time

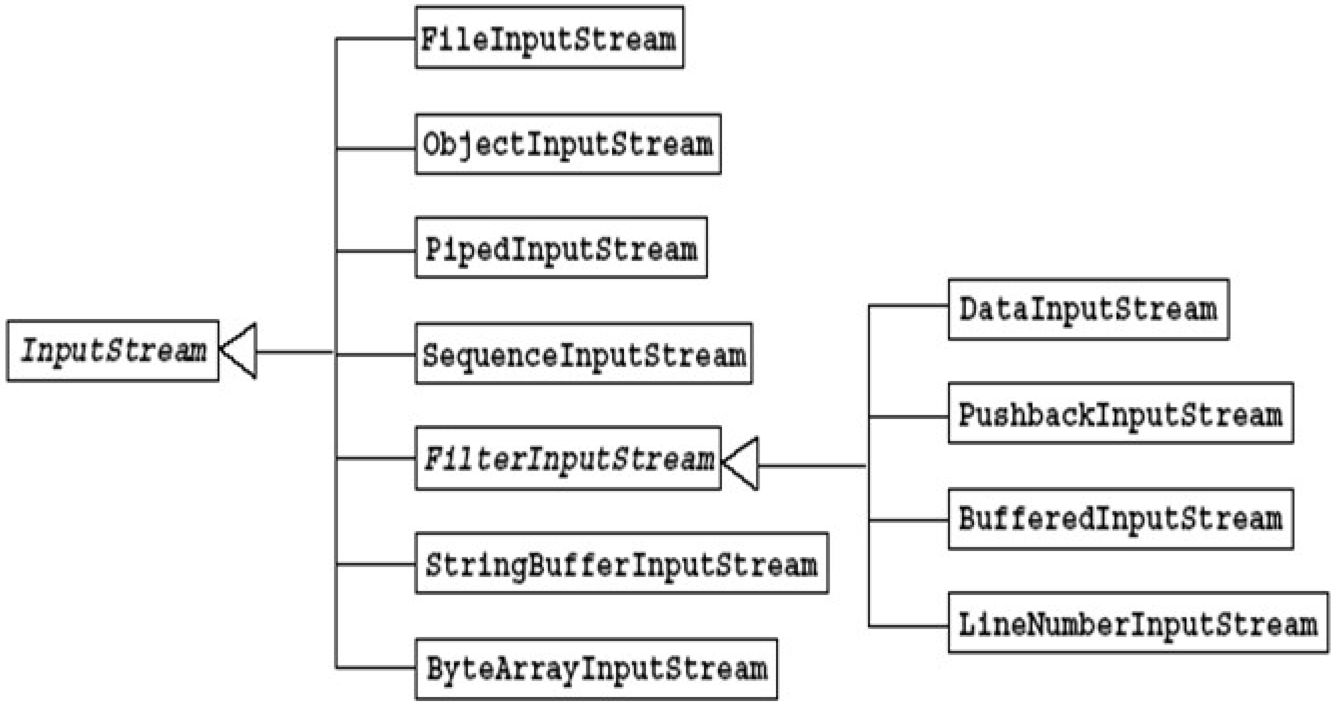


  - A program uses an Output stream to write data **to** a destination, one at a time

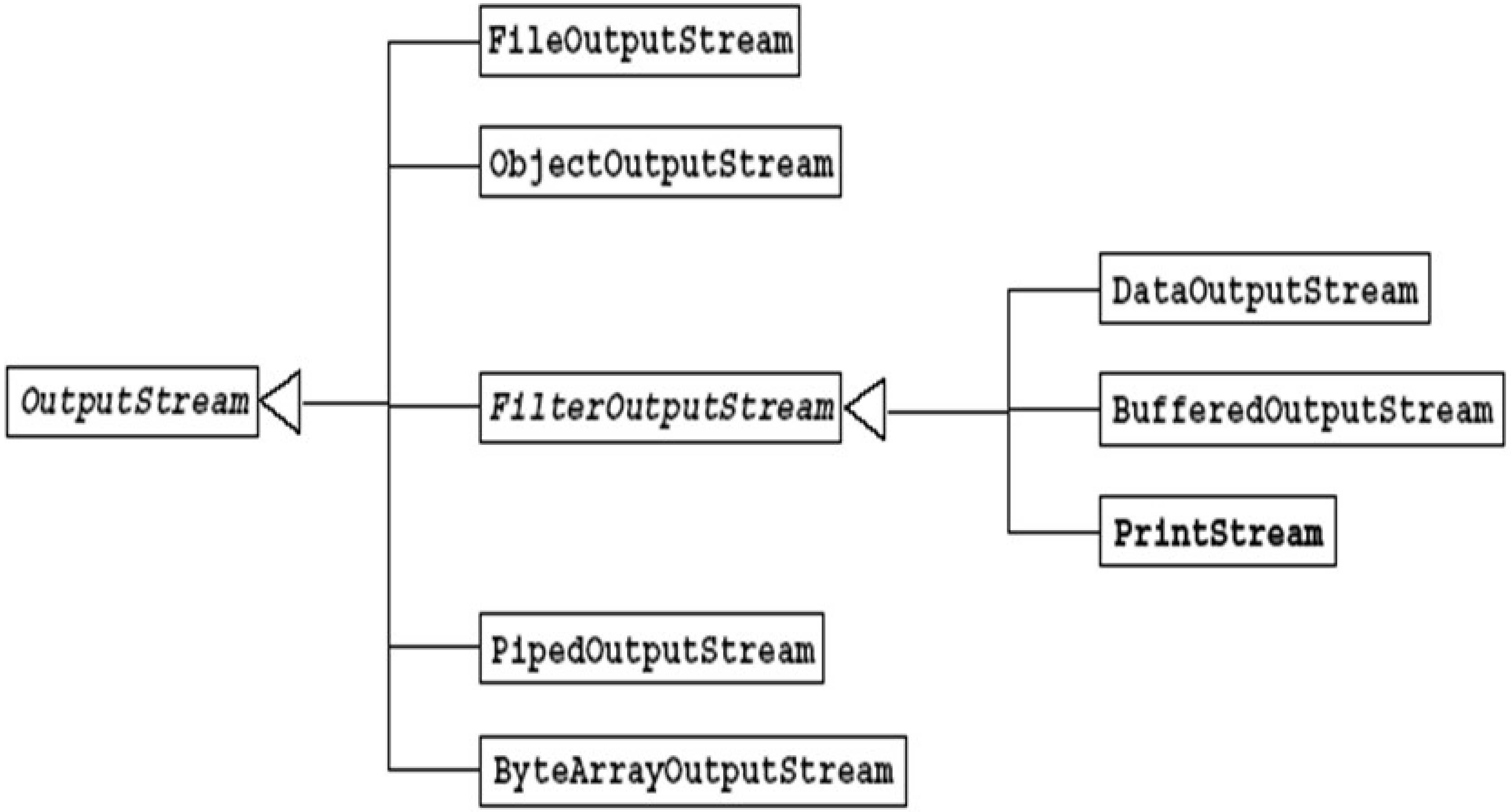


Byte streams

- Programs use **byte streams** to perform input & output of 8 bit bytes
- All *byte stream classes* are descended from InputStream and OutputStream
- Hierarchy of InputStream



Hierarchy of OutputStream



InputStream

- int read()
- int read(byte buf[])
- int read(byte buf[], int offset, int length)
- void close()

Byte streams…

OutputStream

- int write(int c)
- int write(byte buf[])
- int write(byte buf[], int offset, int length)
- void close()
- void flush()

Byte streams…

Example 1

- Write a program using byte streams to copy a text file to another file
- Copy one byte at a time
- Apply FileInputStream & FileOutputStream for the **file** I/O byte streams

Example 1…

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes{
        public static void main(String[] args) throws IOException{
                FileInputStream in=null;
                FileOutputStream out = null;
                try{
                        in = new FileInputStream(“intest.txt”);
                        out = new FileOutputStream(“outtest.txt”);
                        int c;
                        while( (c = in.read()) != -1 ){ //c holds a byte value in its lass 8 bits
                                out.write(c);
                        }
                }finally{
                        if( in !=null) { in.close(); }
                        if ( out !=null ) {  out.close(); }
                }
        }
}
```

When not to use Byte streams

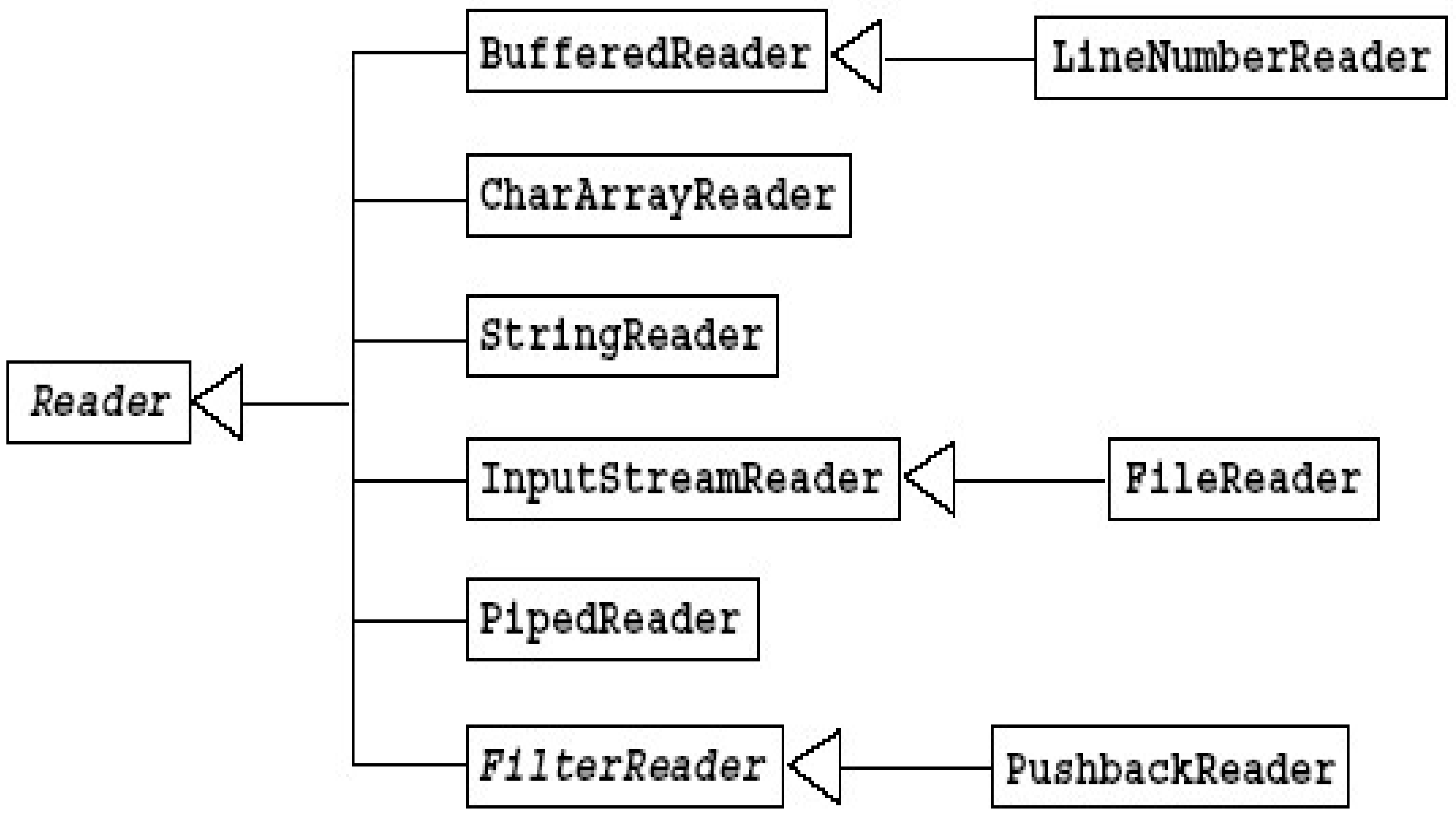
- The previous program represents a kind of low-level I/O that you should avoid
  - Because inTest.txt contain **character data**, the best approach is to use character streams
- Actually, all other stream types are built on byte streams

Character streams

- All character stream classes are descended from **Reader** and **Writer**
- As with byte streams, these are character stream classes that specialize in the **file I/O**
  - **FileReader** and **FileWriter**

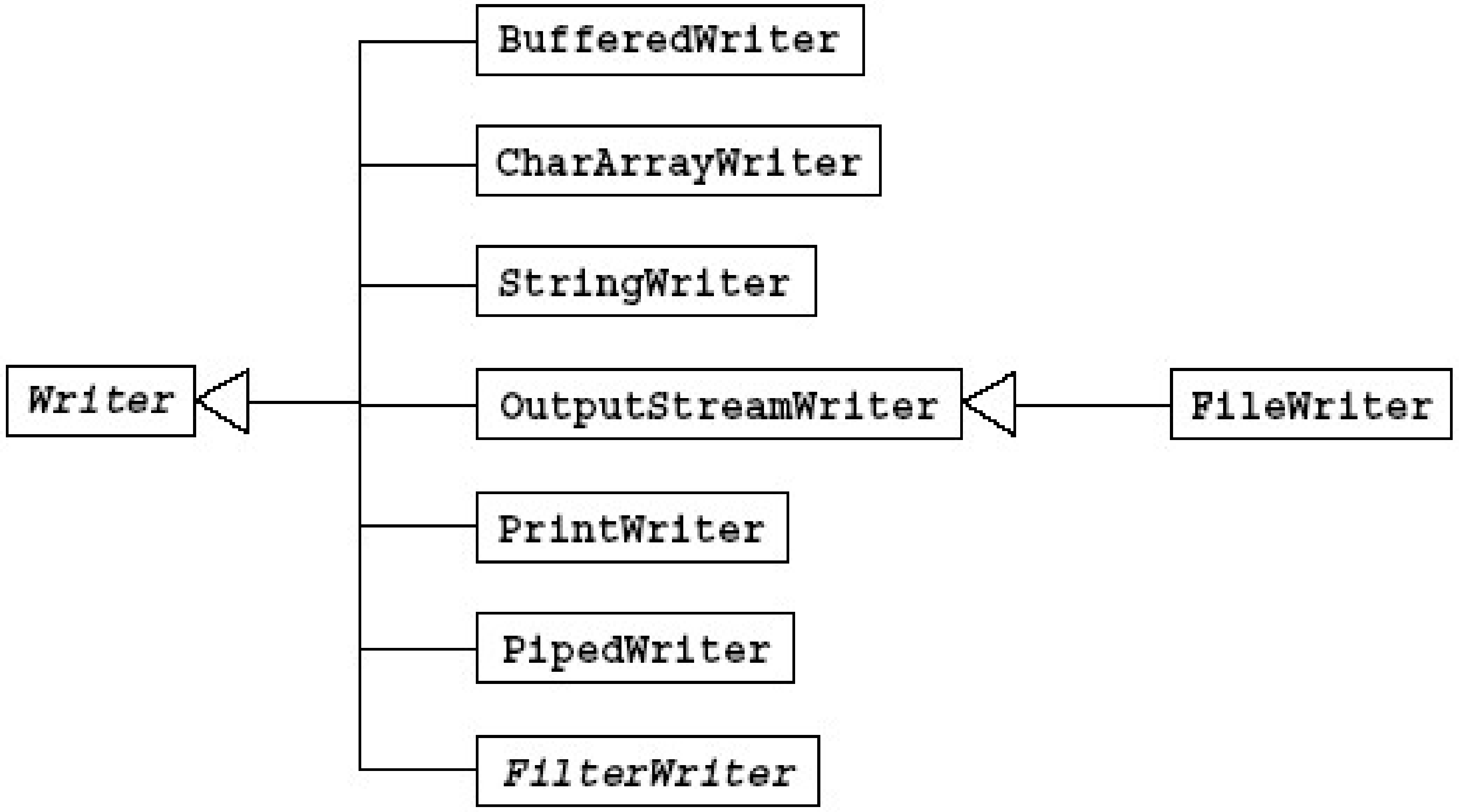
Character streams...

Hierarchy of Reader



Character streams...

Hiearchy of Writer



Character streams...

Reader

- int read()
- int read(char buf[])
- int read(char buf[], int offset, int length)
- void close()

Character streams…

Writer

- int write(int c)
- int write(char buf[])
- int write(char buf[], int offset, int length)
- void close()
- void flush()

Example 2

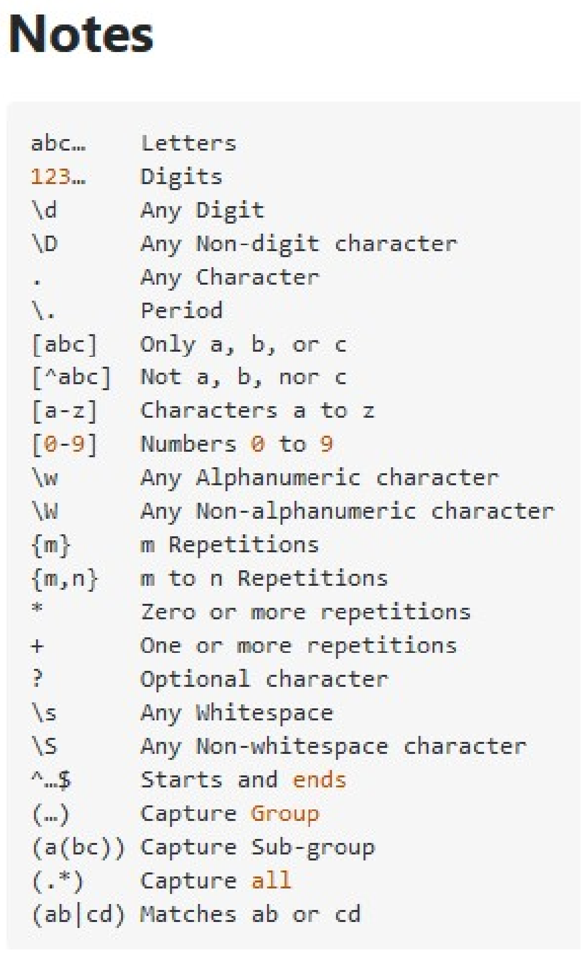
```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CopyCharacters{
        public static void main(String[] args) throws IOException{
                FileReader in = null; FileWriter out = null;
                try{
                        in = new FileReader(“inTest.txt”);
                        out = new FileWriter(“outTest.txt”);
                        char c;
                        while( (c = in.read()) != -1){ //c holds a character value in its last 16 bits
                                out.write(c);
                        }
                }
                finally{
                        if( in !=null ){ in.close(); }
                        if( out !=null) { out.close();}
                }
        }
}
```

Buffered streams

- Examples mentioned use unbuffered I/O, i.e.
  - Each **read** or **write request** is handled **directly** by the underlying OS
  - → make a program much less efficient!
  - Since each request often triggers each disk access, network activity, or some other *relatively expensive operation*
- Java platform implements **buffered I/O streams**
  - Buffered input streams read data from memory area into a buffer
  - Buffered output streams write data to a buffer
- **A program** can convert an unbuffered stream into a buffered stream using the wrapping idiom, e.g., previous example can modified as…
  - in = **new** BufferedReader(**new** FileReader(inTest.txt))
  - out = **new** BufferedWriter(**new** FileWriter(outTest.txt))
- 4 buffered stream classes used to wrap unbuffered streams
  - BufferedInputStream, BufferedOutputStream
  - BufferedReader, BufferedWriter

Scanning

- Objects of type of **Scanner**
  - breaking down formatted input into **tokens**
  - & translating individual tokens according to their data type
- By default, a scanner uses white space (*blanks, tabs, line terminators*) to separate tokens
- To use a different token separator, invoke **useDelimiter()**
  - e.g., use the comma as the token separator
  - s.useDelimiter(",\\s*"); *//Scanner s*



Notes

abc…	Letters
123…	Digits
\d	Any Digit
\D	Any Non-digit character
.	Any Character
\.	Period
[abc]	Only a, b, or c
[^abc]	Not a, b, nor c
[a-z]	Characters a to z
[0-9]	Numbers 0 to 9
\w	Any Alphanumeric character
\W	Any Non-alphanumeric character
{m}	m Repetitions
{m,n}	m to n Repetitions
*	Zero or more repetitions
+	One or more repetitions
?	Optional character
\s	Any Whitespace
\S	Any Non-whitespace character
^…$	Starts and ends
(…)	Capture Group
(a(bc))	Capture Sub-group
(.*)	Capture all
(ab|cd)	Matches ab or cd

Example 1

```
import java.io.*;
import java.util.Scanner;
public class BreakIntoTokens{
        Scanner s = null;
        try{
                s = new Scanner(new BufferedReader(new FileReader(“inTest.txt”)));
                while(s.hasNext()){
                        System.out.println(s.next()); //output individual word  in the file
                }
        }finally{
                if(s !=null){  s.close(); }
        }
}
```

Example 2

```
import java.io.FileReader;  import java.io.BufferedReader;
import java.io.IOException;  import java.util.Scanner; import java.util.Locale
public class TranslateTokens{
        Scanner s = null;
        double sum = 0;
        try{
                s = new Scanner(new BufferedReader(new FileReader("inTest.txt")));
                s.useLocale(Local.US); //separators & decimal symbols are locale specific
                while(s.hasNext()){
                        if(s.hasNextDouble()){  sum += s.nextDouble(); }
                        else {  s.next(); }
                }
        }finally{  if(s !=null){  s.close(); }
        System.out.println(sum);
        }
}
```

I/O from the command line

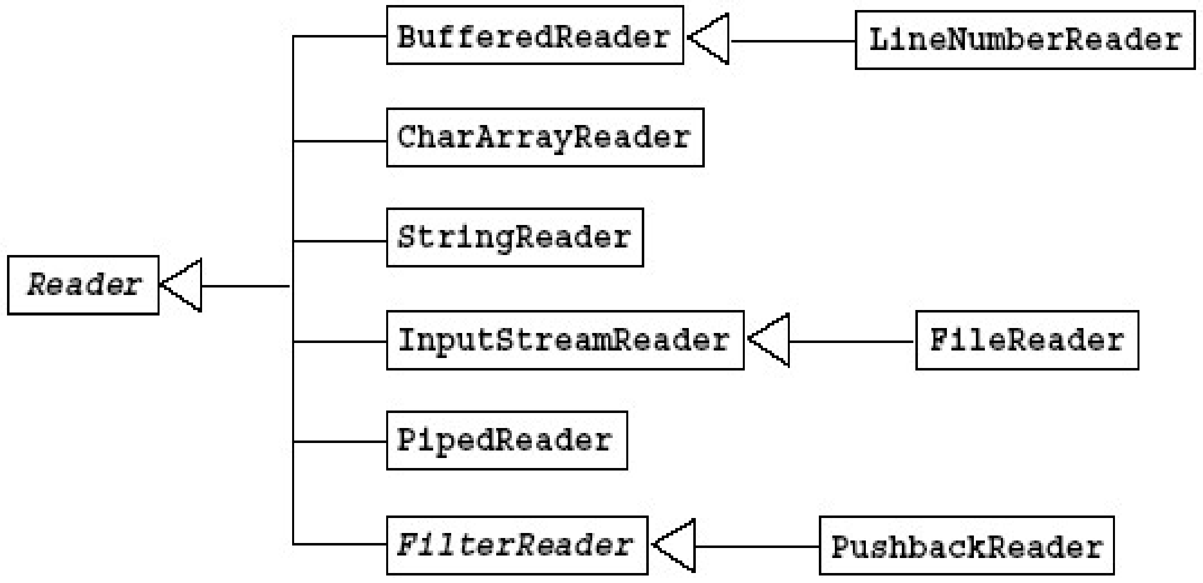
- Java platform support interaction through the command line
    - Standard streams

Standard streams

- Java platform supports 3 standard streams
  - Standard input: accessed through **System.in**
  - Standard output: **System.out**
  - Standard error: **System.err**
- These objects are defined automatically and don't need to be opened
- They are byte streams (not character streams)
  - **System.out** and **System.err** are defined as **PrintStream** objects
- To use standard input as a character stream, wrap **System.in** in InputStreamReader
  - InputStreamReader cin = **new** InputStreamReader(System.in)

Example

```
InputStreamReader reader = new InputStreamReader(System.in);
BufferedReader in = new BufferedReader(reader);

String s;
try {
    s = in.readLine();
}
catch (IOException e) {
...
}
```



File

java.io.File class in java

- **File** class is an abstract representation of file and directory path name
  - *A path name* can be absolute or relative
- File class has several methods for working with files and directories, e.g.,
  - creating **new** files or directories
  - deleting and renaming files or directories
  - listing the **contents** of a directory

E.g. create a file Object

```
try{
    File file = new File("C:/Users/vthnhan/Desktop/test");
    if(file.createNewFile()){....} //new file is created
    else ... //file already exists
}catch(IOException e){ e.printStackTrace() }
```

- Define *an abstract file name* for the **test** file in the directory C:/Users/vthnhan/Desktop/

Constructors

- **File(String** pathname**)**
  - Creates a new File instance by converting *the given pathname string* into an *abstract pathname*.
- **File(File** parent**, String** child**)**
  - Creates a new File instance from a parent abstract pathname and a child pathname string
- **File(String** parent**, String** child**)**
  - Creates a new File instance from a parent pathname string and a child pathname string
- **File(URI** uri**)**
  - Creates a new File instance by converting the given file: URI into an abstract pathname.

Methods

- Files
  - String getName()
  - String getPath()
  - String getAbsolutePath()
  - String getParent()
  - boolean renameTo(File newName)
- Check if the file..
  - boolean exists()
  - boolean canWrite()
  - boolean canRead()
  - boolean isFile()
  - boolean isDirectory()
  - boolean isAbsolute()
- Directory
  - boolean mkdir()
  - String[] list()
- …..

File handling using FileWriter & FileReader

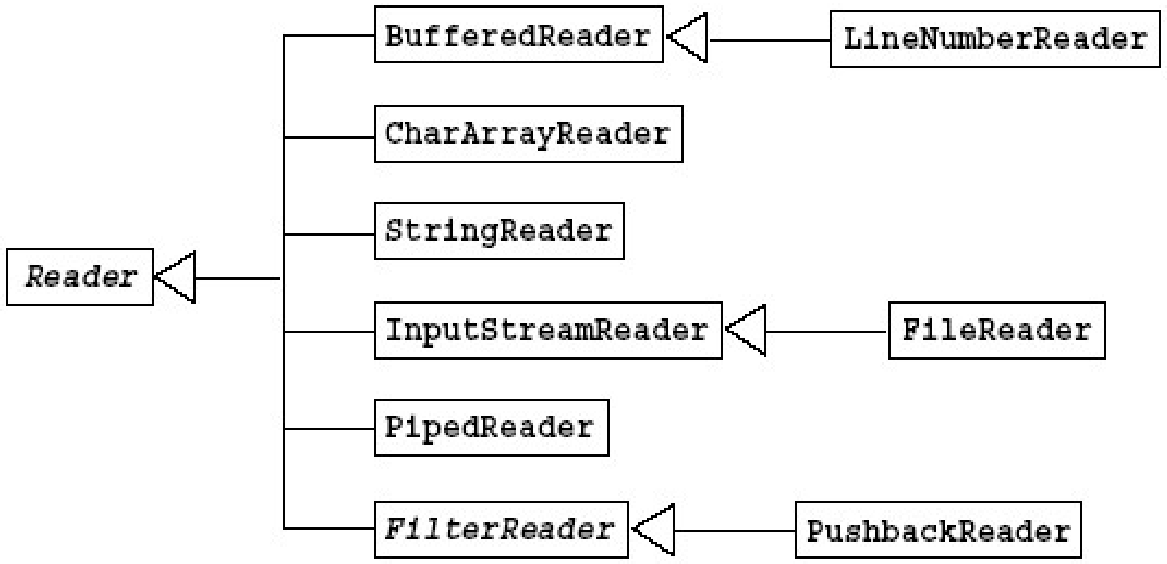
- are used to read and write data from text files
  - they are Character stream classes
- it is recommended not to use FileInputStream & FileOutputStream classes if you have to read and write text as these are Byte stream classes

- **FileWriter** class inherited from OutputStreamWriter class
  - for writing *streams of characters*
  - **BufferedWriter** can be used to improve speed of execution
  - **PrintWriter** used to write a line (with methods print() & println())

- **FileReader** class inherited from InputStreamReader class
  - for reading *streams of characters*
  - BufferedReader can be used (readLine() read a line of text)

E.g 1. read a file

```
File file = new File(”data.txt”);
FileReader reader = new FileReader(file);
BufferedReader in = new BufferedReader(reader);
String s;
try {
    s = in.readLine();
}
catch (IOException e) {
…
}
```



E.g 1. read a file...

```
class Abc {
  public void read(BufferedReader in) {
            String s;
            try {
            s = in.readLine();
                        ...
            }catch (IOException e) {...}
    }
    public void doSomething() {...}
...
}
```

E.g 1. read a file…

```
File file = new File(”data.txt”);
FileReader reader = new FileReader(file);
BufferedReader in = new BufferedReader(reader);

Abc o = new Abc();
o.read(in);
o.doSomething();
```

E.g 2. write text to a file

```
File file = new File(”data.out”);
FileWriter writer = new FileWriter(file);
PrintWriter out = new PrintWriter(writer);
String s = ”Hello”;
try {
    out.println(s);
    out.close();
}
catch (IOException e) {
}
```

E.g 2. write text to a file…

```
class Abc {
…
    public void write(PrintWriter out) {
        …
       try {
           out.println(s);
           out.close();
       }
       catch (IOException e) {…}
    }
}
```

E.g 1. File copy

```
import java.io.*;
public class CopyFile {
    public static void main(String args[]) {
        try {
                FileReader src = new FileReader(args[0]);
                BufferedReader in = new BufferedReader(src);

                FileWriter des = new FileWriter(args[1]);
                PrintWriter out = new PrintWriter(des);
                String s;

                s = in.readLine();
                while (s != null) {
                        out.println(s);
                        s = in.readLine();
                }

                in.close();
                out.close();
        }
        catch (IOException e) { e.printStackTrace();    }
    }
}
```

Sequential access text file

- Read data
  - FileInputStream: read data from a file
  - DataInputStream: read data of primitive data types
  - ObjectInputStream: read objects
- Write data
  - FileOutputStream: write data to a file
  - DataOutputStream: write primitive data
  - ObjectOutputStream: write objects

DataInputStream/DataOutputStream

- **DataInputStream:** read primitive data
  - readBoolean, readByte, readChar, readShort, readInt, readLong, readFloat, readDouble
- **DataOutputStream**: write primitive data
  - writeBoolean, writeByte, writeChar, writeShort, writeInt, writeLong, writeFloat, writeDouble

Write primitive data sequentially

```
import java.io.*;

public class TestDataOutputStream {
    public static void main(String args[]) {
            int a[] = {2, 3, 5, 7, 11};

            try {
                FileOutputStream fout = new FileOutputStream(args[0]);
                DataOutputStream dout = new DataOutputStream(fout);

                for (int i=0; i<a.length; i++)
                        dout.writeInt(a[i]);
                dout.close();
            }
            catch (IOException e) {
                e.printStackTrace();
            }
    }
}
```

Read primitive data sequentially

```
import java.io.*;

public class TestDataInputStream {
    public static void main(String args[]) {

            try {
                FileInputStream fin = new FileInputStream(args[0]);
                DataInputStream din = new DataInputStream(fin);

                while (true) {
                        System.out.println(din.readInt());
                }
            }
            catch (EOFException e) {}
            catch (IOException e) {e.printStackTrace();
            }
    }
}
```

Read/write objects sequentially

- To save a Java object to a database or transfer it over a network
  - We need to convert **the state of an object** into **a byte stream** by using **Serialization**
- To make a java object serializable, we need to implement a marker interface
  - **java.io.Serializable**

Read/write objects sequentially

Example

```
import java.io.Serializable;

class Record implements Serializable {
    private String name;
    private float score;

    public Record(String s, float sc) {
        name = s;
        score = sc;
    }

    public String toString() {
        return "Name: " + name + ", score: " + score;
    }
}
```

Example 1: write objects

```
import java.io.*;

public class TestObjectOutputStream {
    public static void main(String args[]) {
            Record r[] = { new Record("john", 5.0F),
                           new Record("mary", 5.5F),
                           new Record("bob", 4.5F) };

            try {
                FileOutputStream fout = new FileOutputStream(“test.txt”);
                ObjectOutputStream out = new ObjectOutputStream(fout);

                for (int i=0; i<r.length; i++)
                        out.writeObject(r[i]);

                out.close();
            }
            catch (IOException e) {e.printStackTrace();     }
    }
}
```

Example 2: read objects

```
import java.io.*;
public class TestObjectInputStream {
    public static void main(String args[]) {
            Record r;

            try {
                FileInputStream fin = new FileInputStream( “test.txt”);
                ObjectInputStream in = new ObjectInputStream(fin);

                while (true) {
                        r = (Record) in.readObject();
                        System.out.println(r);
                }
            }
            catch (EOFException e) { System.out.println("No more records"); }
            catch (ClassNotFoundException e) {
                System.out.println("Unable to create object");
            }
            catch (IOException e) { e.printStackTrace();    }
    }
}
```

RandomAccessFile class

- is an independent class inherited from the **Object class**
- Support reading and writing data to a file randomly
- Record size **must be fixed**

Example

```
import java.io.*;

public class WriteRandomFile {
    public static void main(String args[]) {
            int a[] = { 2, 3, 5, 7, 11, 13 };

            try {
                File fout = new File(args[0]);
                RandomAccessFile out = new RandomAccessFile(fout, "rw");

                for (int i=0; i<a.length; i++)
                        out.writeInt(a[i]);
                out.close();
            }
            catch (IOException e) {   e.printStackTrace(); }
    }
}
```

Example...

```
import java.io.*;

public class ReadRandomFile {
    public static void main(String args[]) {

            try {
                File fin = new File(args[0]);
                RandomAccessFile in = new RandomAccessFile(fin, "r");

                int recordNum = (int) (in.length() / 4);
                for (int i=recordNum-1; i>=0; i--) {
                        in.seek(i*4);
                        System.out.println(in.readInt()); //read 4 bytes integer
                }
            }
            catch (IOException e) {    e.printStackTrace(); }
    }
}
```

Conclusion

- I/O streams
- Byte streams, character streams
- Sequential and random access files