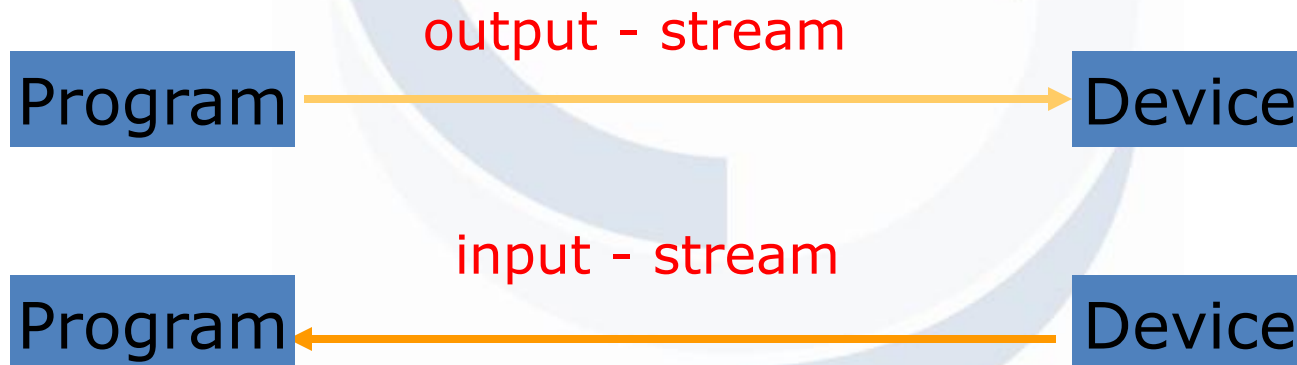




Core Java

# Input Output classes

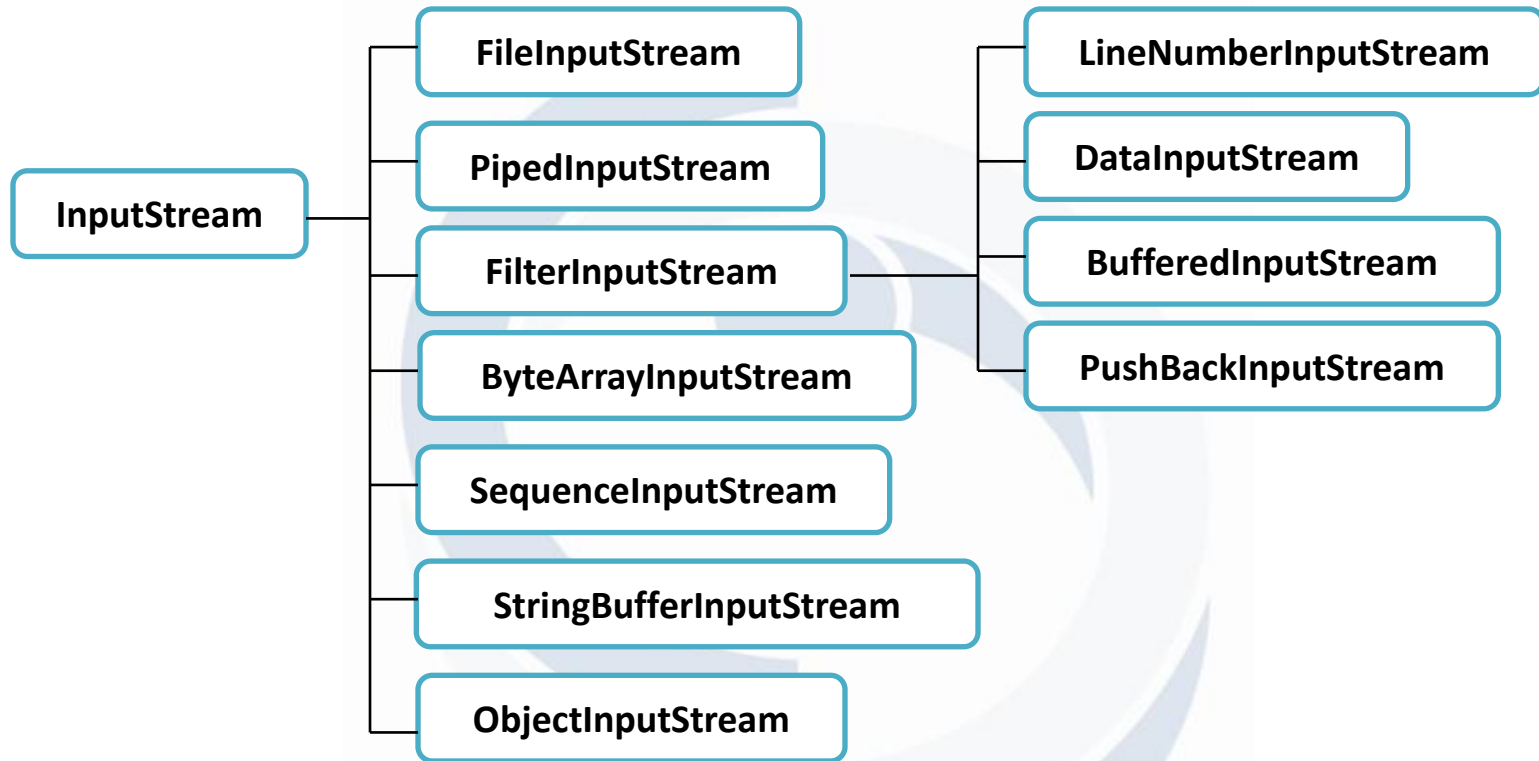
- Java input and output streams are (mostly) supported by the `java.io` package APIs.
  - A stream is a sequence of bytes that flows into (or from) your program from (or into) some kind of information source or device (local file, a socket on the network, variable in memory, another program, a printer, a database).  
Classes provided by package `java.io`
  - Data is transferred to devices by 'streams'



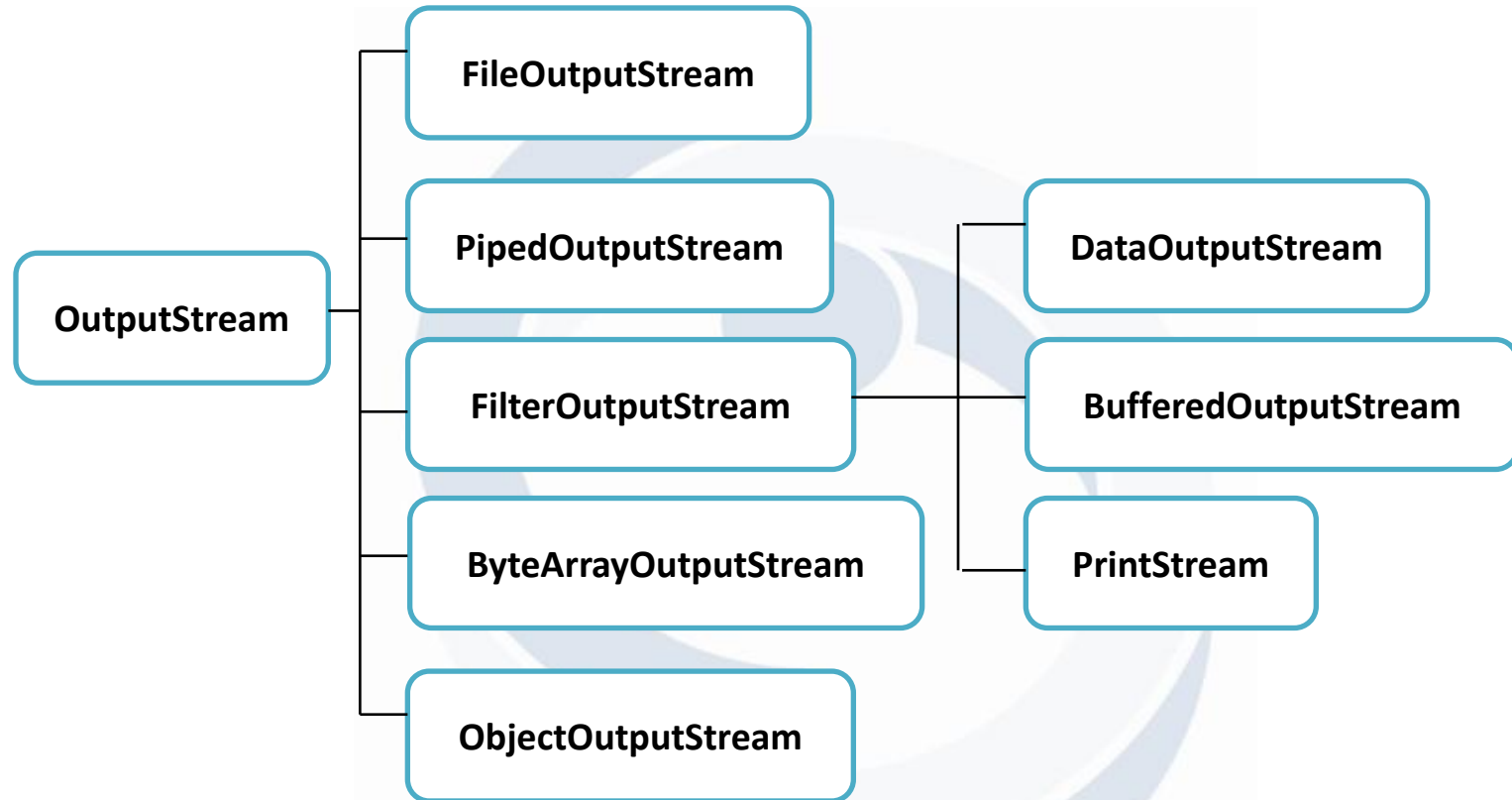
# Stream Class

- **Stream classes are divided into two hierarchies:**
  - One hierarchy deals with character streams while the other deals with byte streams.
  - On the head of the byte stream hierarchy are `InputStream` and `OutputStream` abstract classes, they are responsible for reading and writing 8-bit bytes (binary data).
  - The other hierarchy is headed by the abstract classes `Reader` and `Writer`. They are responsible for reading and writing 16-bit characters (Unicode text).

# InputStream



# OutputStream



# InputStream Class

- **InputStream** is an abstract class, so to truly get some binary data from a file you will need to use one of its concrete sub-classes or extend it yourself.
- **All classes that are derived of it inherit the following methods:**
  - **`public abstract int read() throws IOException`**
    - The method is abstract and must be implemented by the subclasses. The method should handle reading one byte of data from the stream and return it as an **`int`**.
  - **`public int read(byte[] b) throws IOException`**
    - Reads some number of bytes from the input stream and stores them into the buffer array **`b`**. The number of bytes actually read is returned as an **`int`**.

# InputStream Methods

- **public int read(byte[] b, int off, int len) throws IOException**
  - The method reads bytes into the array buffer starting at element **off**. It will attempt to read up to **len** bytes or until the end of the stream is reached. It will return the number of bytes that were read as an **int**.
- **public long skip(long n) throws IOException**
  - Skips over and discards n bytes of data from this input stream or until the end of the stream is reached. It is useful for skipping over parts of the file which you don't wish to process. Number of bytes skipped is returned as **long**
- **public void close() throws IOException**
  - Closes this input stream and releases any system resources associated with the stream.

# OutputStream Class

- **This abstract class is the superclass of all classes representing an output stream of bytes.**
  - Classes that extend **OutputStream** are typically used for writing images and sounds (binary) data.
- **The class contains the following methods:**
  - **public abstract void write(int b) throws IOException**
    - Writes the specified byte to this output stream. Subclasses of **OutputStream** must provide an implementation for this method.
  - **public void write(byte[] b) throws IOException**
    - Writes b.length bytes from the specified byte array to this output stream.



# OutputStream Methods

- `public void write(byte[] b,int off,int len) throws IOException`
  - Writes **len** bytes from the specified byte array starting at offset **off** to this output stream.
- `public void flush() throws IOException`
  - Flushes this output stream and forces any buffered output bytes to be written out.
- `public void close() throws IOException`
  - Closes this output stream and releases any system resources associated with this stream.

# FileInputStream and FileOutputStream

- A **FileInputStream** obtains input bytes from a file in a local file system.
- A **FileOutputStream** is an output stream for writing data to a local File.
  - Each time we invoke these methods the program access the hard disk.
  - Both classes permit the transfer of a very small amount of data at each operation. Handling your data this way is extremely inefficient.
  - The following programs uses **FileInputStream** and **FileOutputStream** for reading a file and copying its content to another file.

# Example

```
1  import java.io.*;
2  public class CopyFile
3  {
4      public static void main(String[] args) {
5          try
6          {
7              long lStart = System.currentTimeMillis();
8              FileInputStream in  = new FileInputStream("source.txt");
9              FileOutputStream out = new FileOutputStream("dest.txt");
10             int c;
11             while ((c = in.read()) != -1) // Read/Write a byte
12                 out.write(c);
13
14             in.close();
15             out.close();
16             long lEnd = System.currentTimeMillis();
17             System.out.println("Total time (milisec) = "
18                               +(lEnd - lStart));
19         }
20     }
```

# Exception Methods

```
catch (IOException e)  {  
    System.out.println("**An IO problem occurred " + e);  
} catch (IOException e){  
    System.out.println("**An IO problem occurred "+ e);  
}  
  
} //main  
  
} //class
```

# Buffer

- We can easily improve the file copying process introduced on the previous slide by using **BufferedInputStream** and **BufferedOutputStream**. These classes are sub-classing **FilterInputStream** and **FilterOutputStream** and adding the functionality of managing buffers.
  - **BufferedInputStream**
    - When the **BufferedInputStream** is created, an internal buffer array is created. As bytes from the stream are read, the internal buffer is refilled as necessary from the contained input stream, many bytes at a time. Every read operation invoked does not force an access the hard disk, but rather uses the data placed inside the buffer.

# BufferedOutputStream

- `BufferedOutputStream`.
  - By setting up such an output stream, an application can write bytes to the underlying output stream without necessarily causing a call to the hard disk for each byte written. The data is written into an internal buffer. It is written to the underlying stream. When the buffer reaches its capacity, the buffer output stream is closed, or the buffer output stream is explicitly flushed.

# BufferedInputStream and BufferedOutputStream

```
InputStream in = new FileInputStream("x.txt");
```

```
OutputStream out = new FileOutputStream("newx.txt");
```

```
BufferedInputStream bufIn = new BufferedInputStream(in);
```

```
BufferedOutputStream bufOut = new BufferedOutputStream(out) ;
```

# Example

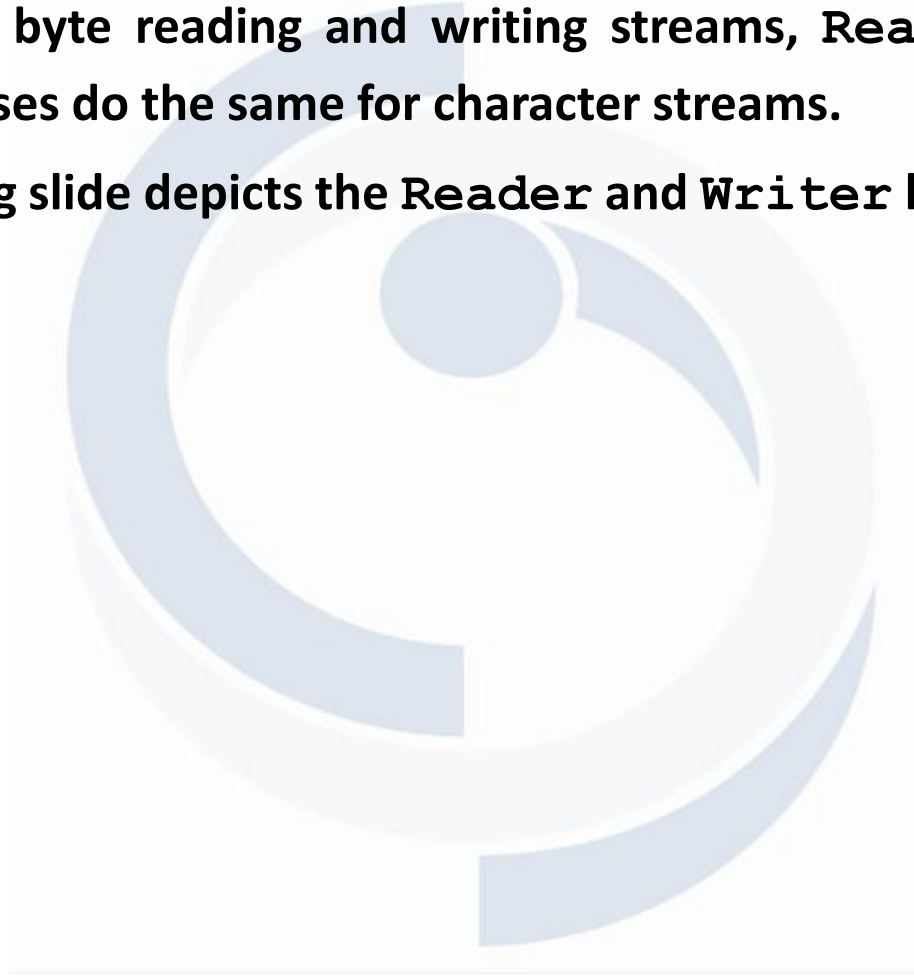
```
import java.io.*;
public class ImprovedCopyFile{
    public static void main(String[] args) {
        try {
            InputStream in = new FileInputStream("Source.txt");
            OutputStream out = new FileOutputStream("test.txt");
            BufferedInputStream bufIn = new BufferedInputStream(in);
            BufferedOutputStream bufOut = new BufferedOutputStream(out);
            int c;
            while ((c = bufIn.read()) != -1)
                bufOut.write(c) ;
        }catch (IOException io){
            System.err.println("**An IO problem has occurred " + io);
        }finally{
            //close file
        }

    }
}
```

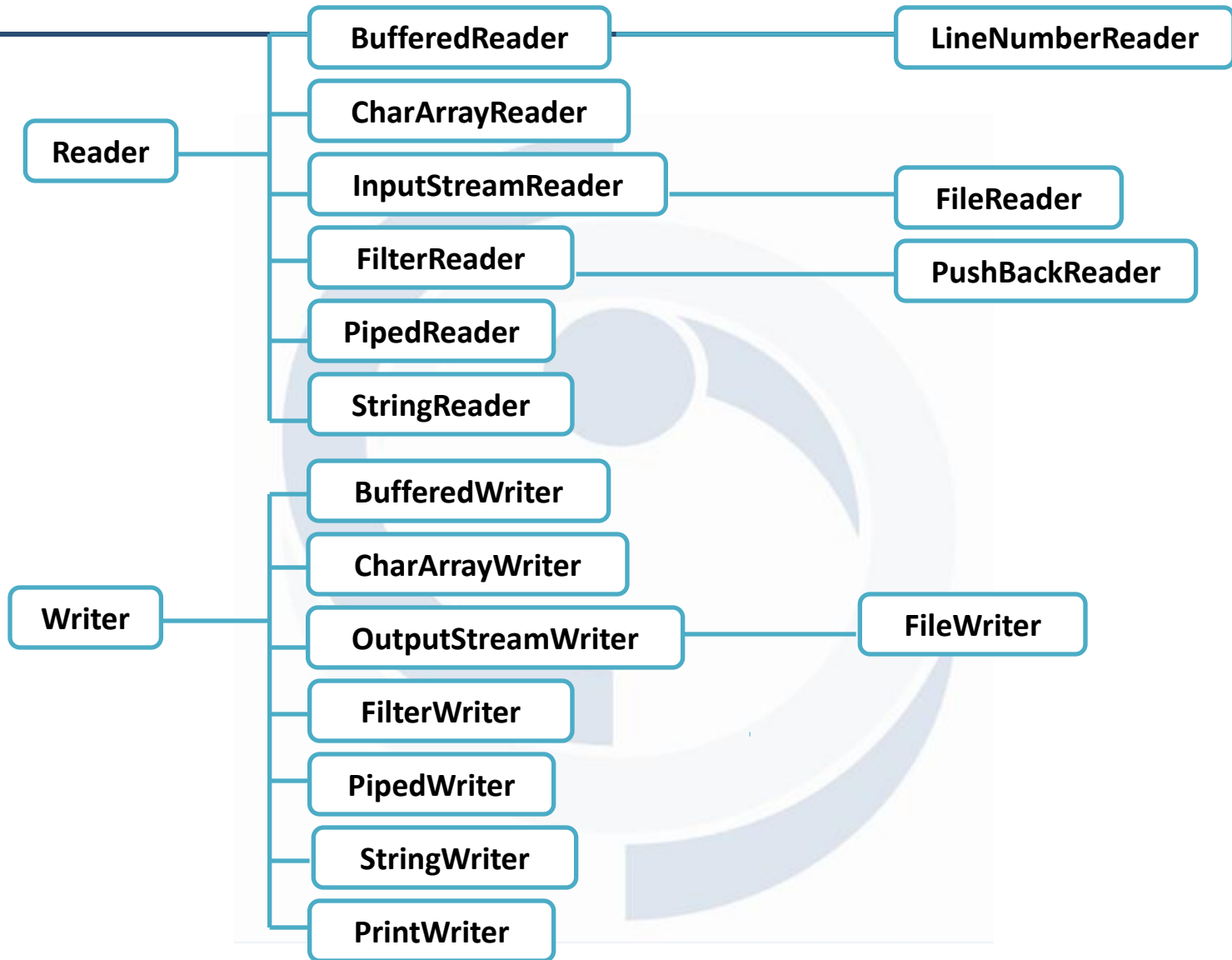


# Character Stream

- Just like `InputStream` and `OutputStream` , abstract classes head a hierarchy of byte reading and writing streams, `Reader` and `Writer` abstract classes do the same for character streams.
- The following slide depicts the `Reader` and `Writer` hierarchies



# Character Stream



# Example

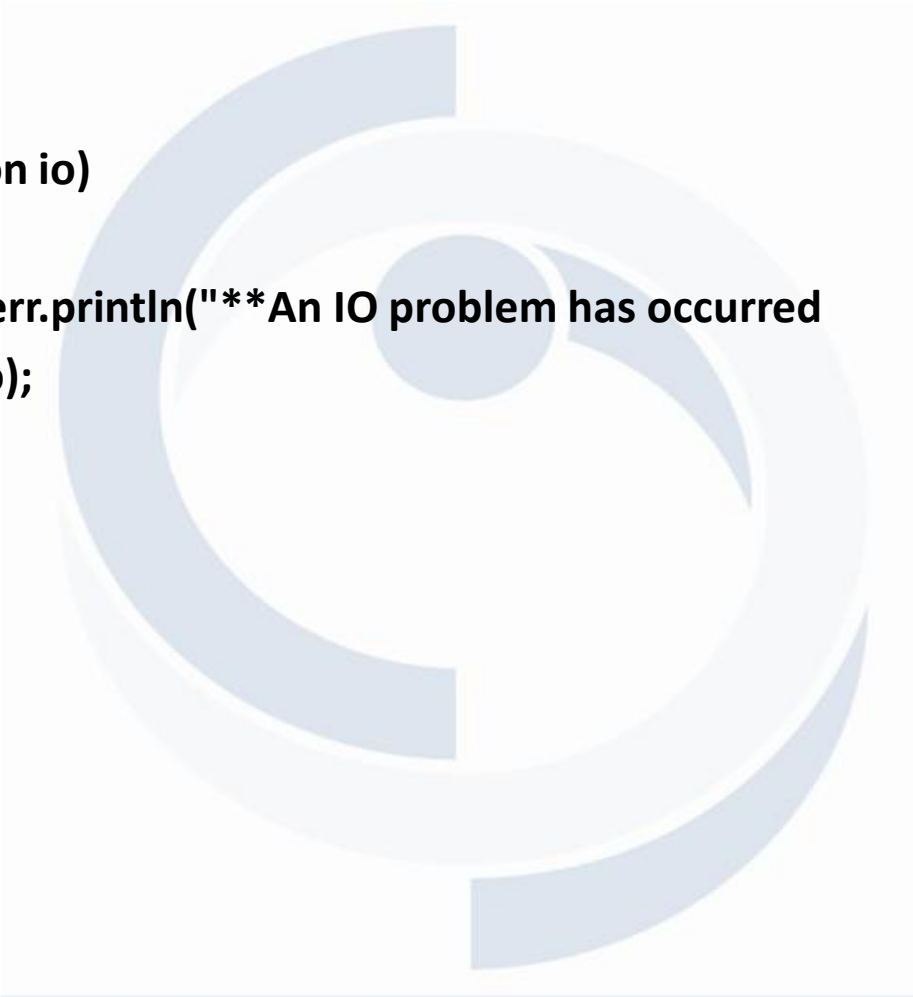
```
import java.io.*;

public class CopyCharFile{
    public static void main(String[] args) {
    try{
        |
        FileReader in  = new FileReader("sample.txt");
        FileWriter out = new FileWriter("newsample.txt");
        int ch;

        while ((ch = in.read()) != -1)
            out.write(ch);
    }
```

# Example

```
in.close();  
out.close();  
}  
catch (IOException io)  
{  
    System.err.println("**An IO problem has occurred  
        " + io);  
}  
}  
}
```



# CopyBufStream.java

```
1  import java.io.*;
2  public class CopyBufStream
3  {
4      public static void main(String[] args)
5      {
6          try
7          {
8              long lStart = System.currentTimeMillis();
9              FileReader in = new FileReader(args[0]);
10             FileWriter out = new FileWriter(args[1]);
11             BufferedReader bufIn = new BufferedReader(in);
12             BufferedWriter bufOut = new BufferedWriter(out);
13             String line;
14             // Read and write a whole line
15             while ((line = bufIn.readLine()) != null) {
16                 bufOut.write(line);
17                 bufOut.newLine();
18             }
```



## Again, Closing up

```
19  bufIn.close();
20  bufOut.close();
21  long lEnd = System.currentTimeMillis();
22  System.out.println("Total time (milisec) = " +
23                      (lEnd - lStart));
24  }
25  catch (IOException io)
26  {
27      System.err.println("**An IO problem has occurred
28                          " + io);
29  }
30  }
31 }
```



# read and readLine

- **Reading from `BufferedReader` can be accomplished using the following methods:**
  - Reading a single character:
    - `public int read() throws IOException`
  - Reading a buffer:
    - `public int read(char[] cbuf, int off, int len) throws IOException`
  - Reading a line of text:
    - `public String readLine() throws IOException`



# Console I/O

- **User and application interaction is accomplished using console I/O.**
  - Writing to the "standard output", the console that launched the application, is achieved using **System.out**
    - **System.out** is of type **PrintStream**, so it allows writing to the standard output a variety of data types using either **println()** or **print()**:
      - **void println(boolean)**
      - **void println(char)**
      - **void println(String)**
      - **void println(float)**
      - **void println(int)**





# Reading From the Standard Input

- **System.in**, which is an **InputStream** object, accepts user keyboard information and transfers it to the application.
  - **InputStream** reads binary data, so some bridging must be done to convert it to buffered character data.

```
InputStreamReader stdin = new InputStreamReader(System.in);
```

```
BufferedReader bufRead = new BufferedReader(stdin);
```

**InputStreamReader** wraps **System.in**, converting bytes into Unicode

**BufferedReader** allows the program to read line after line from the standard input.



# ReadKeyboard.java

```
1  import java.io.*;
2  class ReadKeyboard
3  {
4      public static void main(String args[])  {
5          InputStreamReader stdin = new
6          InputStreamReader(System.in);
7          BufferedReader bufRead  = new
8          BufferedReader(stdin);
9          while(true)
10         {
11             try {
12                 System.out.println(bufRead.readLine());
13             }
14             catch(IOException io) {
15                 io.printStackTrace();
16             }
17         }
18     }
19 }
```



# Object Streams

- So far, we have dealt with binary streams and character streams, this section focuses on object streams.
- Using objects streams allows us to conveniently store objects to external files and extracts the objects from those files.
  - The process of storing and retrieving is called **serialization**.
    - Writing an object to a file is referred to as serializing the object.
      - Class `ObjectOutputStream` supports this process.
    - Reading an object from a file is referred to as **deserializing** the object.
      - Class `ObjectInputStream` supports this process.



# How to Serialize

- **In order for an object to be transferred through a stream it must follow some rules.**
  - First, it should implement the **Serializable** interface.
    - The serialization interface is a markup interface, it has no methods or fields and serves only to identify the semantics of being serializable.
  - Second, the class must be declared public.
  - Third, the class data members should also implement **Serializable** or be declared **transient**.
    - **transient** will be discussed in a little while.
  - Forth, if the class has a super-class this too must implement **Serializable**, if it does not, it must supply a default constructor.
- **When an object is being written to a stream all its data members are recursively written as well.**



# Input and Output Object Stream

- **ObjectOutputStream** receives in its constructor a **FileOutputStream** as an argument:

- `ObjectOutputStream outObj = new ObjectOutputStream(new`
  - `FileOutputStream("obj.file"));`

- **Writing an object is done by the method:**

- `public final void writeObject(Object obj) throws`
  - `IOException`

- **ObjectInputStream** receives in its constructor a **FileInputStream** as an argument:

- `ObjectInputStream inObj = new ObjectInputStream(new`
  - `FileInputStream("obj.file"));`

- **Reading an object is done by the method:**

- `public final Object readObject()`
  - `throws OptionalDataException,`
  - `ClassNotFoundException, IOException`



# Serialization Example

```
import java.io.*;
public class Student implements Serializable
{
    int rollno;
    String name;
    int marks;
    Student(int rollno,String name,int marks) {
        this.rollno=rollno;
        this.name=name;
        this.marks=marks;
    }
    public void display(){

        System.out.println(" "+rollno+" Name "+name+" "+marks );
    }
}
```



# Serialization Example – cont'd

```
import java.io.*;
class SerializationDemo
Public static void main(String arg[]) {
    FileOutputStream out = new FileOutputStream("file.txt");
    ObjectOutputStream objOut = new ObjectOutputStream(out);
    Student s1= new Student(1,"Anita",70);
    objOut.writeObject(s1); // Write to file

    FileInputStream in = new FileInputStream("file.txt");
    ObjectInputStream objIn = new ObjectInputStream(in);
    Student s2 = (Student)objIn.readObject(); // Read from file
    s2.display();
}
} // End of class
```



# Transient Data Members

- **There are two reasons for declaring a data member as transient:**
  - The member is an object that does not implement **Serializable**.
  - We don't wish to write the data member value into the stream.
- **Current time and date are examples for values that should be recalculated rather than saved.**

```
public class ReadWriteObj implements Serializable {  
    transient private Date m_date = new Date;  
}
```

**When an object of this type is retrieved, the member `m_date` will have a null value**

