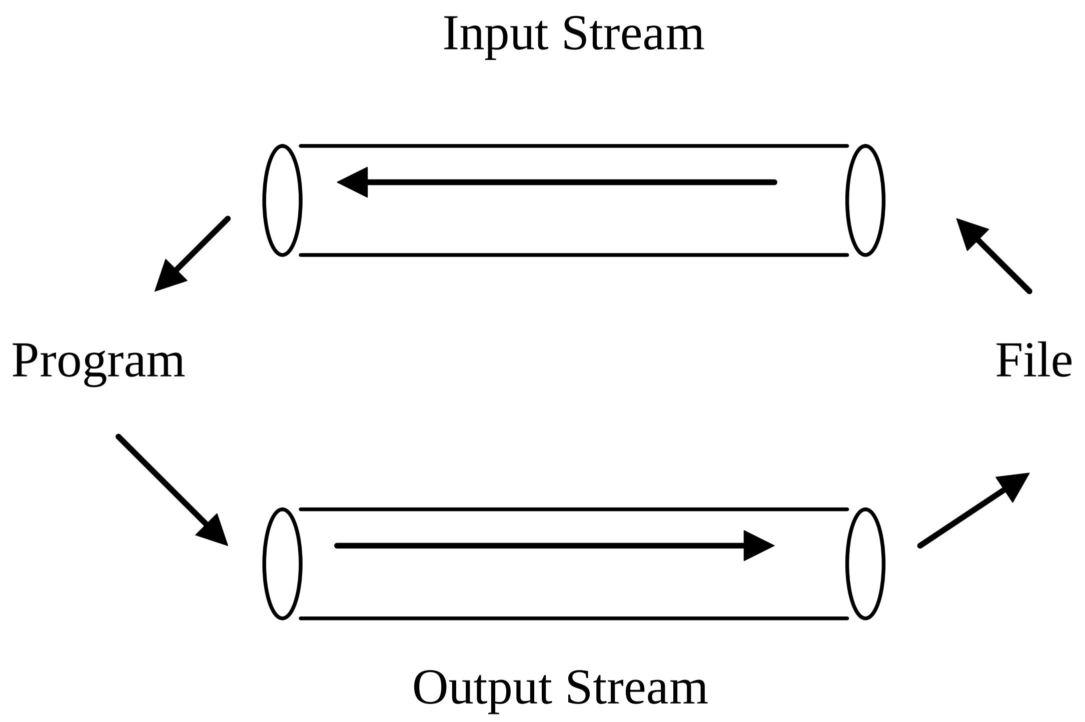
FILES AND STREAMS

By Pakita Shamoi

STREAMS

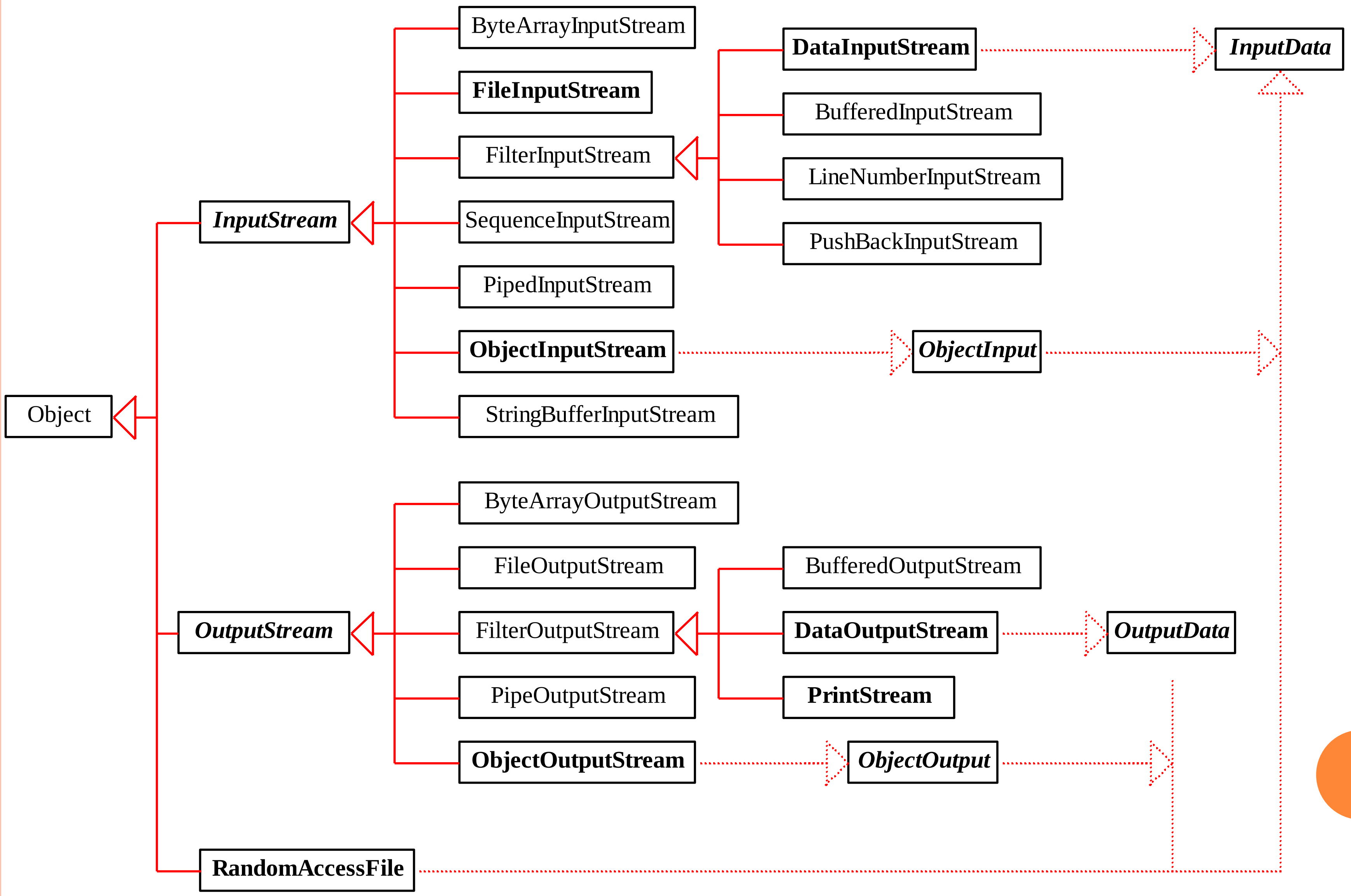
- A *stream* is an abstraction of the continuous one-way flow of data.
  You can think of it as of an ordered sequences of data that have a ***source*** (input streams) or a ***destination*** (output streams)



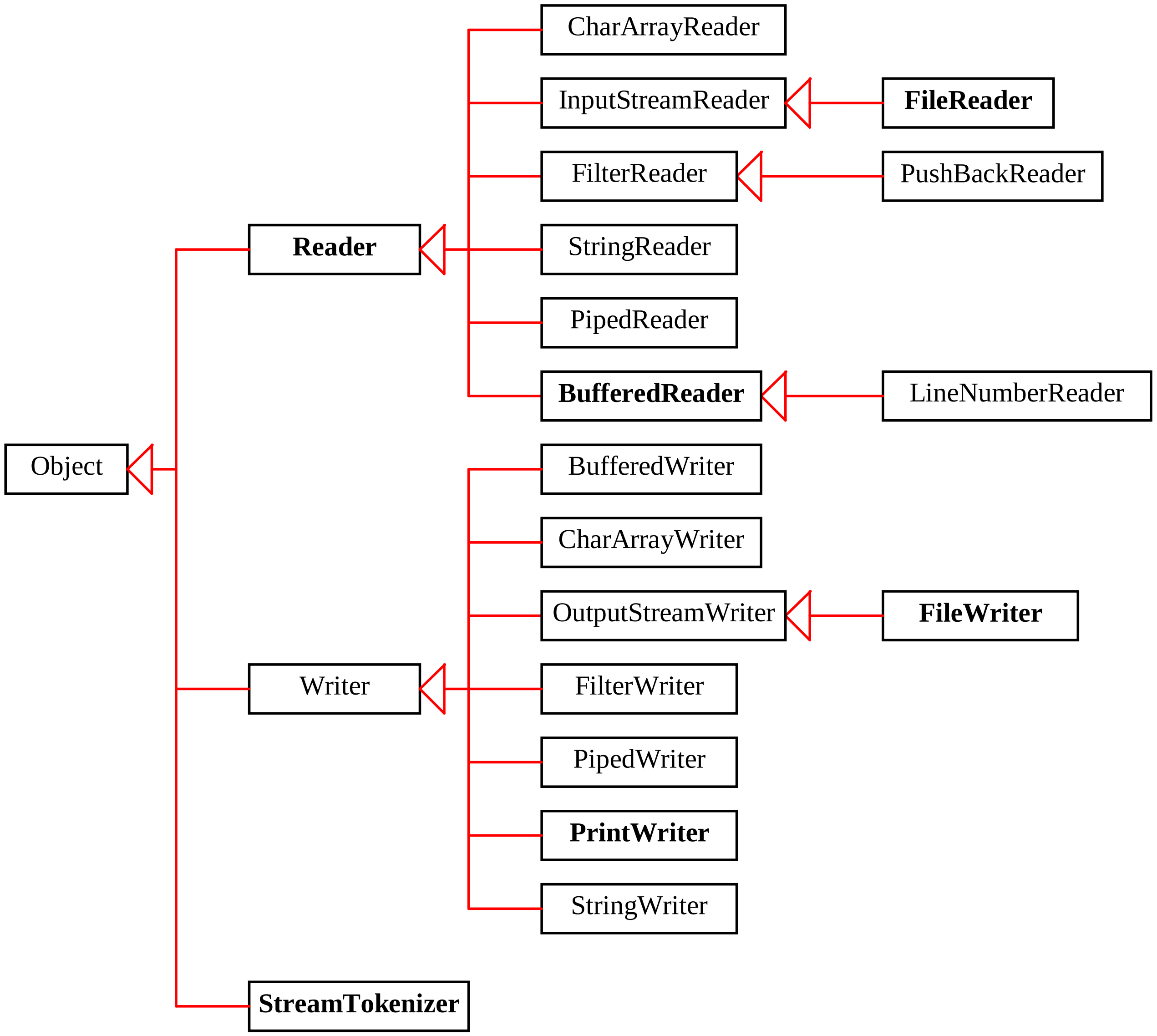
Stream Classes

- The stream classes can be categorized into two types: *byte streams* and *character streams*.
- The `InputStream/OutputStream` class is the root of all byte stream classes, and the `Reader/Writer` class is the root of all character stream classes. The subclasses of `InputStream/OutputStream` are analogous to the subclasses of `Reader/Writer`.

BYTE STREAM CLASSES



Character Stream Classes



Processing External Files

You must use file streams to read from or write to a disk file. You can use `FileInputStream` or `FileOutputStream` for byte streams, and you can use `FileReader` or `FileWriter` for character streams.

File I/O Stream Constructors

Constructing instances of `FileInputStream`, `FileOutputStream`, `FileReader`, and `FileWriter` from file names:

```
FileInputStream infile = new FileInputStream("in.dat");

FileOutputStream outfile = new FileOutputStream("out.dat");

FileReader infile = new FileReader("in.dat");

FileWriter outfile = new FileWriter("out.dat");
```

DATA STREAMS

The data streams (`DataInputStream` and `DataOutputStream`) read and write Java primitive types in a machine-independent fashion, which enables you to write a data file in one machine and read it on another machine that has a different operating system or file structure.

DataInputStream & DataOutputStream Methods

- DataInputStream
  - int readShort() throws IOException
  - int readInt() throws IOException
  - int readLong() throws IOException
  - float readFloat() throws IOException
  - double readDouble() throws IOException
  - char readChar() throws IOException
- DataOutputStream
  - void writeByte(byte b) throws IOException
  - void writeInt(int i) throws IOException
  - void writeLong(long l) throws IOException
  - void writeDouble(double d) throws IOException
  - void writeChar(char c) throws IOException
  - void writeBoolean(boolean b) throws IOException
  - void writeBytes(String l) throws IOException

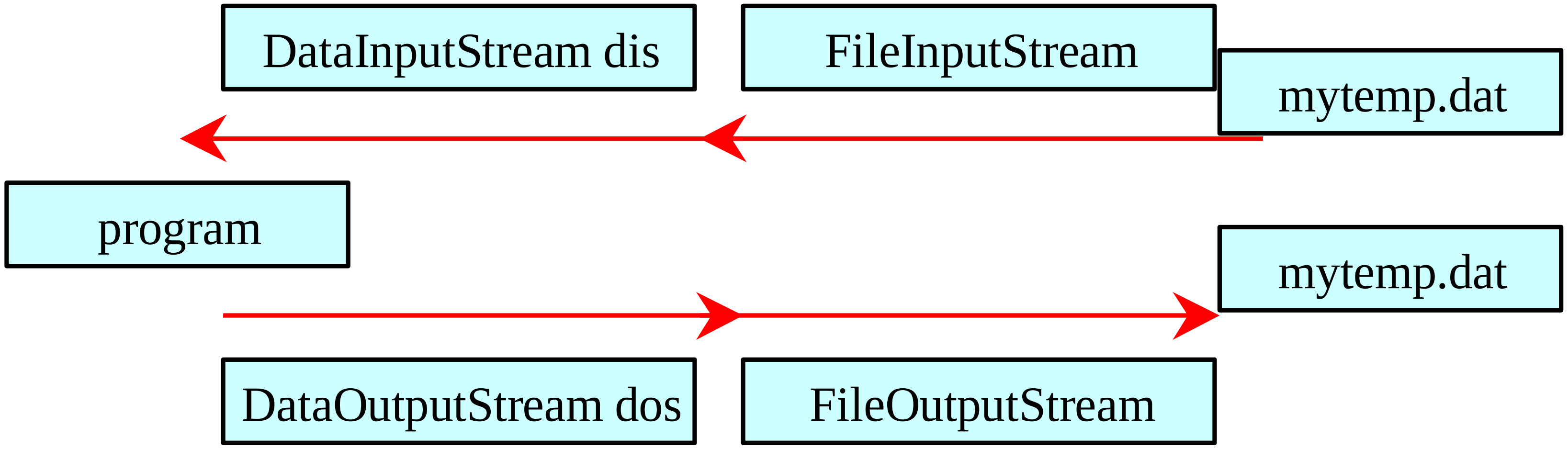
Data I/O Stream Constructors

- `DataInputStream infile = new DataInputStream(new FileInputStream("in.dat"));`

  Creates an input file for in.dat.

- `DataOutputStream outfile = new DataOutputStream(new FileOutputStream("out.dat"));`

  Creates an output file for out.dat.



Print Streams

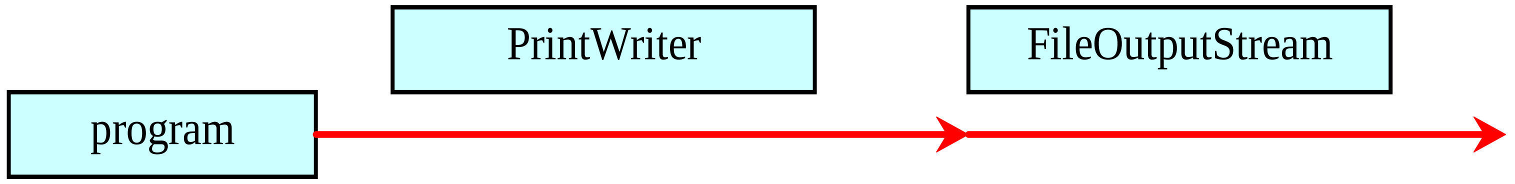
The data output stream outputs a binary represen-tation of data, so you cannot view its contents as text. In Java, you can use print streams to output data into files. These files can be viewed as text.

The **`PrintWriter`** classes provide this functionality.

```
void print(String s)
void print(char c)
void print(char[] cArray)
void print(int i)
void print(long l)
void print(double d)
void print(boolean b)
```

PrintWriter Constructors

- `PrintWriter(Writer out)`
- `PrintWriter(Writer out, boolean autoFlush)`
- `PrintWriter(OutputStream out)`
- `PrintWriter(OutputStream out, boolean autoFlush)`



Buffered Streams

Java introduces buffered streams that speed up input and output by reducing the number of reads and writes. In the case of input, *a bunch of data is read all at once instead of one byte at a time*. In the case of output, *data are first cached into a buffer*, then written all together to the file.

Using buffered streams is highly recommended.

- `BufferedInputStream(InputStream in)`
- `BufferedOutputStream(OutputStream in)`
- `BufferedReader(Reader in)`
- `BufferedWriter(Writer out)`

Add more efficiency

- `So,` **`BufferedReader`** reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.

```
BufferedReader (Reader in)
```

- For example:
  - ✧ to wrap an `InputStreamReader` inside a `BufferedReader`

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

  - ✧ to wrap a `FileReader` inside a `BufferedReader`

```
BufferedReader in
         = new BufferedReader(new FileReader("fileName"));
```

then you can invoke `in.readLine()` to read from the file line by line

```
import java.io.*;
public class EfficientReader {
    public static void main (String[] args) {
       try {
          BufferedReader br = new BufferedReader(new FileReader(“a.txt”));

          // get line
          String line = br.readLine();
          // while not end of file… keep reading and displaying lines
          while (line != null) {
                System.out.println("Read a line:");
                System.out.println(line);
                line = br.readLine();
          }
          // close stream
          br.close();
       } catch(FileNotFoundException fe) {
              System.out.println("File not found: “+ args[0]");
       } catch(IOException ioe) {
              System.out.println("Can’t read from file: “+args[0]);
       } }}
```

Object Streams

- Object streams enable you to perform input and output at the object level.
- To enable an object to be read or write, the object's defining class has to implement the *java.io.Serializable*

- The *Serializable* interface is a *marker interface*. It has no methods, so you don't need to add additional code in your class that implements *Serializable*.

- Implementing this interface enables the Java serialization mechanism to automate the process of storing the objects and arrays.

THE OBJECT STREAMS

You need to use :

- The `ObjectOutputStream` class for **storing objects** (writing them)

- The `ObjectInputStream` class for **restoring objects** (reading them)

SERIALIZATION & DESERIALIZATION EXAMPLE

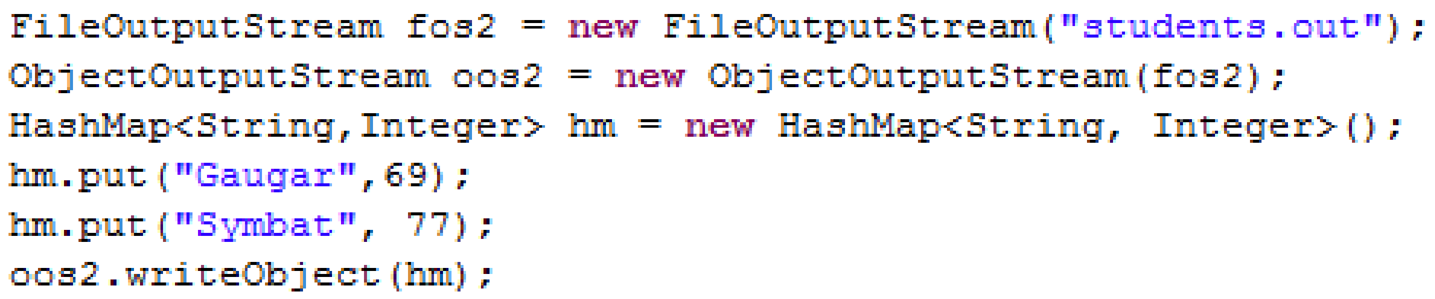
- Serialization

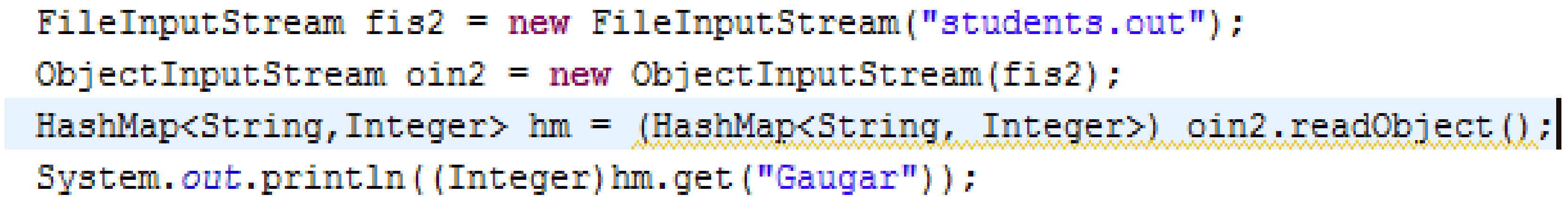
```java
FileOutputStream fos = new FileOutputStream("book.out");
ObjectOutputStream oos = new ObjectOutputStream(fos);
Book b = new Book(220,"Ann Karenina");
oos.writeObject(b);
oos.flush();
oos.close();
```

- Deserialization

```java
FileInputStream fis = new FileInputStream("book.out");
ObjectInputStream oin = new ObjectInputStream(fis);
Book b = (Book) oin.readObject();
System.out.println(b);
```

‘Object’ can be object of objects


```
FileOutputStream fos2 = new FileOutputStream("students.out");
ObjectOutputStream oos2 = new ObjectOutputStream(fos2);
HashMap<String,Integer> hm = new HashMap<String, Integer>();
hm.put("Gaugar",69);
hm.put("Symbat", 77);
oos2.writeObject(hm);
```


```
FileInputStream fis2 = new FileInputStream("students.out");
ObjectInputStream oin2 = new ObjectInputStream(fis2);
HashMap<String,Integer> hm = (HashMap<String, Integer>) oin2.readObject();
System.out.println((Integer)hm.get("Gaugar"));
```

Working with Files

- Sequential-Access file: the `File` streams - `FileInputStream`, `FileOutputStream`, `FileReader` and `FileWriter`—allow you to treat a file as a stream to input or output sequentially
  - Each file stream type has the following constructors:
    - A constructor that takes a `String` which is the name of the file
    - A constructor that take a `File` object which refers to the file

- Random-Access file: **`RandomAccessFile`** allows you to **read/write** data beginning at the a **specified location**
  - a ***file pointer*** is used to guide the starting position

Random Access Files

- So, Java provides the **`RandomAccessFile`** class to allow a file to be read and updated at the same time.

- It includes typical methods, like `readInt()`, `readLong()`, `writeDouble()`, `readLine()`, `writeInt()`, and `writeLong()`.

- **`void seek(long pos)`**

  Sets the pointer to where the next read or write need to happen

- **`long getFilePointer()`**

  Returns the current pointer offset, in bytes, from the beginning of the file

- **`long length()`** - Returns the length of the file.

- **`final void writeBytes(String s)`** - Writes a string to the file .

Example of RandomAccessFile

```
import java.io.*;
class Filecopy {
   public static void main(String args[]) {
      RandomAccessFile f1 = null;
      RandomAccessFile f2 = null;
      long filesize = -1;
      byte[] buffer1;
      try {
         f1 = new RandomAccessFile("a.txt", "r");
         f2 = new RandomAccessFile("b.txt", "rw");
      } catch (FileNotFoundException e) {
         System.out.println("File not found");
         System.exit(100);
      }
      try {
         filesize = f1.length();
         int bufsize = (int)filesize/2;
         buffer1 = new byte[bufsize];
         f1.readFully(buffer1, 0, bufsize);
         f2.write(buffer1, 0, bufsize);
      } catch (IOException e) {
         System.out.println("IO error occurred!");
         System.exit(200);
      }
   }
}
```

THE FILE CLASS

- The **`File`** class is particularly useful for retrieving information about a file or a directory from a disk.
  - A `File` object actually **represents a path**, not necessarily an underlying file
  - A `File` object doesn't open files or provide any file-processing capabilities
- Constructors:
  - `public File( String name)`
  - `public File( File directory, String name)`
- Main methods
  - `boolean canRead() /  boolean canWrite()`
  - `boolean exists()`
  - `boolean isFile() / boolean isDirectory()`
  - `String getPath()`
  - `String getParent()`
  - `String getName()`
  - `long length()`

OVERVIES OF KEY TERMS

- Stream
- Input streams, output streams
- Reading from file, writing to file
- Object streams
- RandomAccessFile
- File class
- BufferedReader and BufferedWriter
- Data streams
- Print streams

Main topics out here (for your project):

- Reading from file, writing to file
- Object streams
  - **Serialization**
  - **Deserialization**
- *RandomAccessFile (?)*
- BufferedReader and BufferedWriter
- Print streams

Overview

- **Character stream** is useful when we want to process text files.
- **A byte stream** is suitable for processing raw data like binary files.
- Names of character streams typically end with Reader/Writer and names of byte streams end with InputStream/OutputStream