




Application Program Development

Segment : Java Files and Streams

Mahboob Ali




Agenda for Week 9

- Lecture
 - Java input and output streams
 - Java binary reading and writing
 - Java objects reading and writing
 - Lab
 - Part 1 : In-Lab - Design and create a GUI based Application
 - Part 2 : DIY - using the GUI designed in-lab to write events
- 

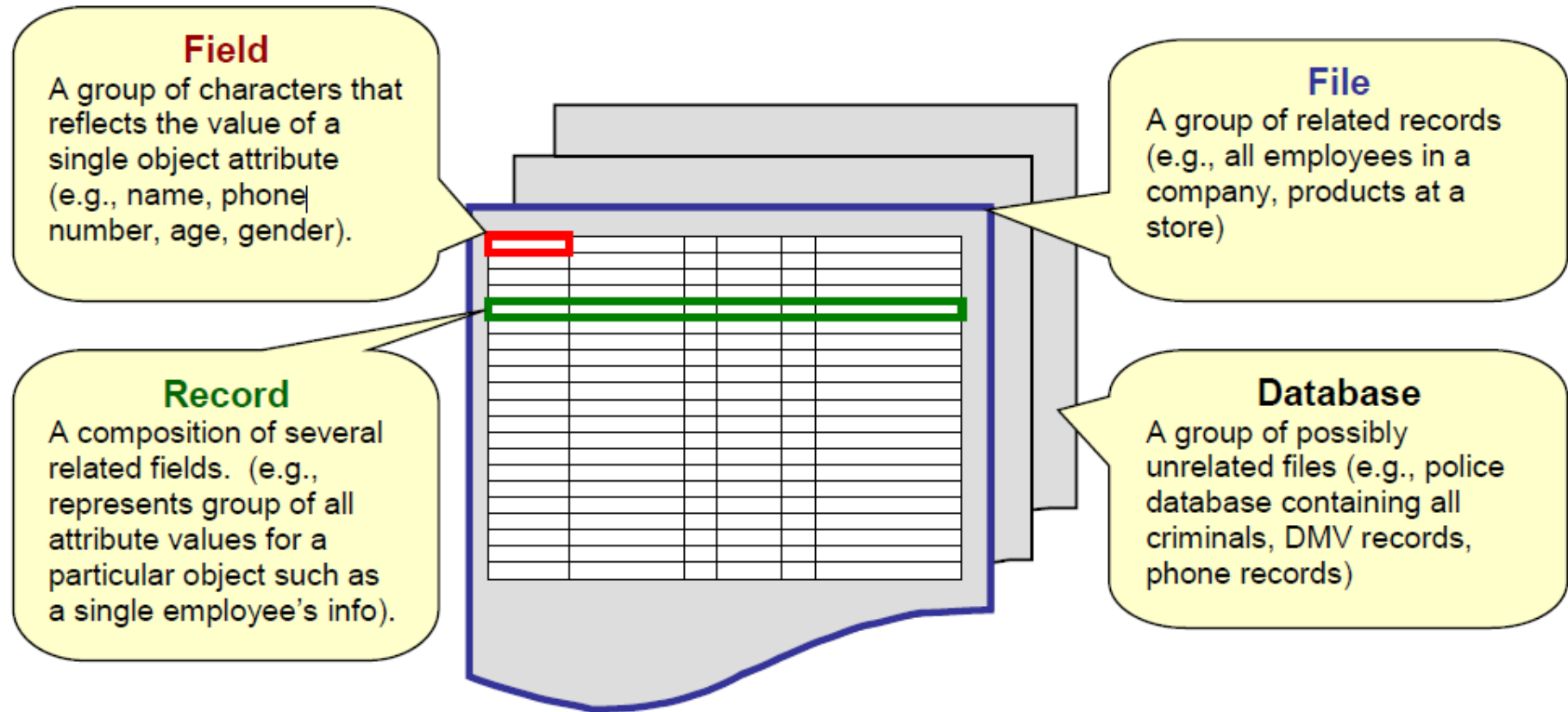


Outcomes

- Examine Input/ Output classes in java
 - Create and use I/O streams in java
 - Differences between Byte, Character and Buffered Streams
 - Designing and developing File I/O programs.
 - Using 3rd party libraries for reading and writing data
- 

Files and Streams

- File processing is very important since eventually, all data must be stored externally from the machine so that it will not be erased when the power is turned off.
- Here are some of the terms related to file processing:



Files and Streams

- In JAVA, we can store information from our various objects by extracting their attributes and saving these to the file.
- To use a file, it must be first **opened**.
- When done with a file, it MUST be **closed**.
- We use the terms **read** to denote getting information *from* a file and **write** to denote saving information *to* a file.
- The contents of a file is ultimately reduced to a set of numbers from **0** to **255** called **bytes**.
- In JAVA, files are represented as **Stream** objects.
- The idea is that data “streams” (or flows) to/from the file ... similar to the idea of streaming video that you may have seen online.
- Streams are *objects* that allow us to send or receive information in the form of bytes.
- The information that is put into a stream, comes out in the same order.

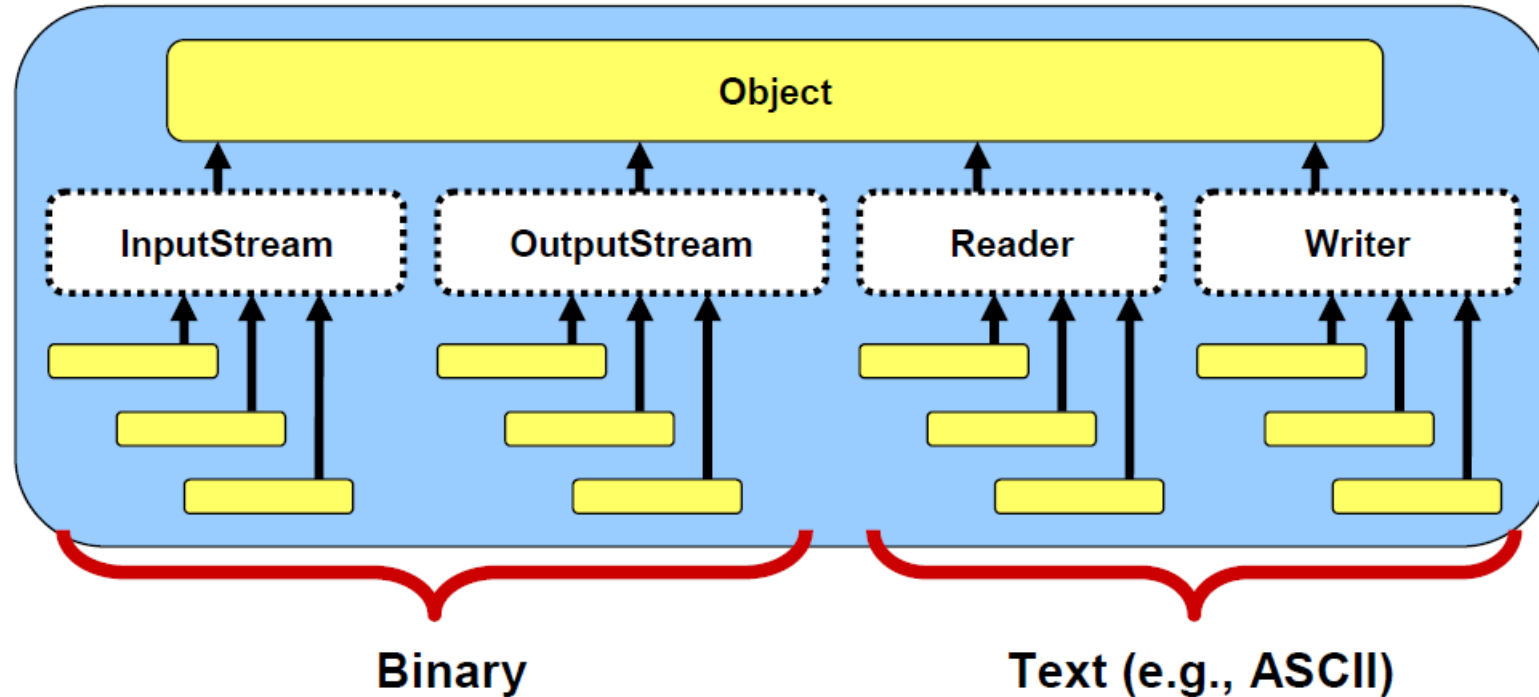
Streams

- **Streams** are actually very general in that they provide a way to send or receive information to and from:

- ❖ files
 - ❖ networks
 - ❖ different programs
 - ❖ any I/O (i.e., input/output) devices (e.g., console and keyboard)
-
- When we first start executing a JAVA program, 3 streams are automatically created:
 - `System.in` // for inputting data from keyboard
 - `System.out` // for outputting data to screen
 - `System.err` // for outputting error messages to screen or elsewhere
 - In fact, there are many stream-related classes in JAVA.
 - We will look at a few and how they are used to do file I/O. The various **Streams** differ in the way that data is “entered into” and “extracted from” the stream.
 - As with **Exceptions**, **Streams** are organized into different hierarchies.

Streams

- JAVA contains four main stream-related hierarchies for transferring data as **binary** bytes or as **text** bytes:



- It is interesting to note that there is no common **Stream** class from which these main classes inherit.
- Instead, these 4 **abstract** classes are the root of more specific subclass hierarchies.
- A rather large number of classes are provided by JAVA to construct streams with the desired properties.

I / O

- Typically I/O is a bottleneck in many applications.
- It is very time consuming to do I/O operations when compared to internal operations.
- For this reason, **buffers** are used.
- Buffered output allows data to be collected before it is actually sent to the output device.
- Only when the buffer gets full does the data actually get sent.
- This reduces the amount of actual output operations.
- (**Note:** The **flush()** command can be sent to buffered streams in order to empty the buffer and cause the data to be sent "immediately" to the output device. Input data can also be buffered.)

I / O

- By the way, what is **System.in** and **System.out** exactly ? We can determine their respective classes with the following code:

```
System.out.print("System.in is an instance of ");  
System.out.println(System.in.getClass()); System.out.print("System.out is  
an instance of ");  
System.out.println(System.out.getClass());
```

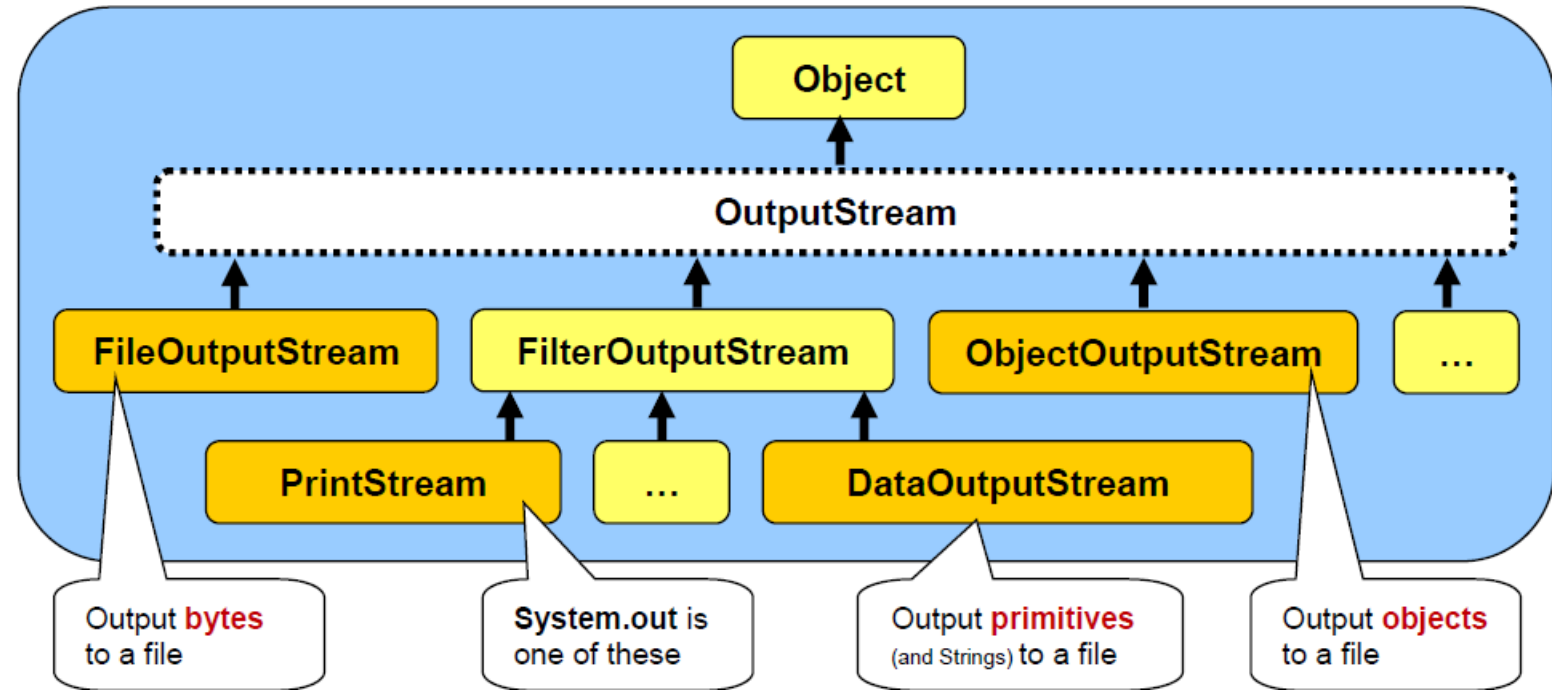
- This code produces the following output:

```
System.in is an instance of class java.io.BufferedReader  
System.out is an instance of class java.io.PrintStream
```

- So we have been using these streams for displaying information and getting information from the user through the keyboard.
- We will now look at how the classes are arranged in the different stream sub-hierarchies.

Reading and Writing Binary Data

- Following the sub-hierarchy of JAVA'S **OutputStream**



- The streams in this sub-hierarchy are responsible for outputting binary data (i.e., data in a form that is not readable by text editors).
- OutputStreams** have a **write()** method that allows us to output a single byte of data at a time.

Reading and Writing Binary Data

- To open a file for binary writing, we can create an instance of **FileOutputStream** using one of the following constructors:

```
new FileOutputStream(String fileName);  
new FileOutputStream(String fileName, boolean append);
```

- The first constructor opens a new "file output stream" with that name.
- If one exists already with that name, it is overwritten (i.e., erased).
- The second constructor allows you to determine whether you want an existing file to be overwritten or appended to.
- *If the file does not exist, a new one with the given name is created.*
- *If the file already exists prior to opening, then the following rules apply:*
 - if append = **false** the existing file's contents is discarded and the file will be overwritten.
 - if append = **true** the new data to be written to the file is appended to the end of the file.

Reading and Writing Binary Data

- We can output simple bytes to a **FileOutputStream** by using the **write()** method, which takes a single byte (i.e., a number from **0** to **255**) as follows:

```
FileOutputStream out;  
out = new FileOutputStream("myFile.dat");  
out.write('H');  
out.write(69);  
out.write(76);  
out.write('L');  
out.write('O');  
out.write('!');  
out.close();
```

- This code outputs the characters **HELLO!** to a file called **"myFile.dat"**.
- The file will be created (if not existing already) in the **current** directory/folder (i.e., the directory/folder that your JAVA program was run from).
- Alternatively, you can specify where to create the file by specifying the whole path name instead of just the file name as follows:

```
FileOutputStream out;  
out = new FileOutputStream("F:\\My Documents\\myFile.dat");
```

- Notice the use of "two" backslash characters within the String constant (because the backslash character is a special character which requires it to be preceded by a backslash ... just as **\n** is used to create a new line).

Reading and Writing Binary Data

- Using this strategy, we can output either characters or positive **integers** in the range from **0** to **255**.
- Notice in the code that we closed the file stream when done. This is important to ensure that the operating system (e.g., Windows 10) releases the “file handles” correctly.
- When working with files in this way, two exceptions may occur:
- opening a file for reading or writing may generate a **java.io.FileNotFoundException**
- reading or writing to/from a file may generate a **java.io.IOException**
- You should handle these exceptions with appropriate **try/catch** blocks:

(Full program is on next slide)

Reading and Writing Binary Data

```
import java.io.*; // Need to import since all Streams are in this package

public class FileOutputStreamTestProgram {
    public static void main(String[] args) {
        try {
            FileOutputStream out;
            out = new FileOutputStream("myFile.dat");
            out.write('H');
            out.write(69);
            out.write(76);
            out.write('L');
            out.write('O');
            out.write('!');
            out.close();
        } catch (FileNotFoundException e) {
            System.out.println("Error: Cannot open file for writing");
        } catch (IOException e) {
            System.out.println("Error: Cannot write to file");
        }
    }
}
```

Reading and Writing Binary Data

- The code in previous slide allows us to output any data as long as it is in **byte** format.
- This can be tedious.
- For example, if we want to output the integer **7293901** ... we have a few choices:
 - break up the integer into its **7** digits and output these digits one at a time (very tedious)
 - output the **4** bytes corresponding to the integer itself (recall: an **int** is stored as **4** bytes)
- *Either way, these are not fun.*
- Fortunately, JAVA provides a **DataOutputStream** class which allows us to output whole primitives (e.g., **ints, floats, doubles**) as well as whole **Strings** to a file!

Reading and Writing Binary Data (DataOutputStream)

- The **DataOutputStream** acts as a “wrapper” class around the **FileOutputStream**. It takes care of breaking our primitive data types and Strings into separate bytes to be sent to the **FileOutputStream**.
- There are methods to write each of the primitives as well as **Strings**:

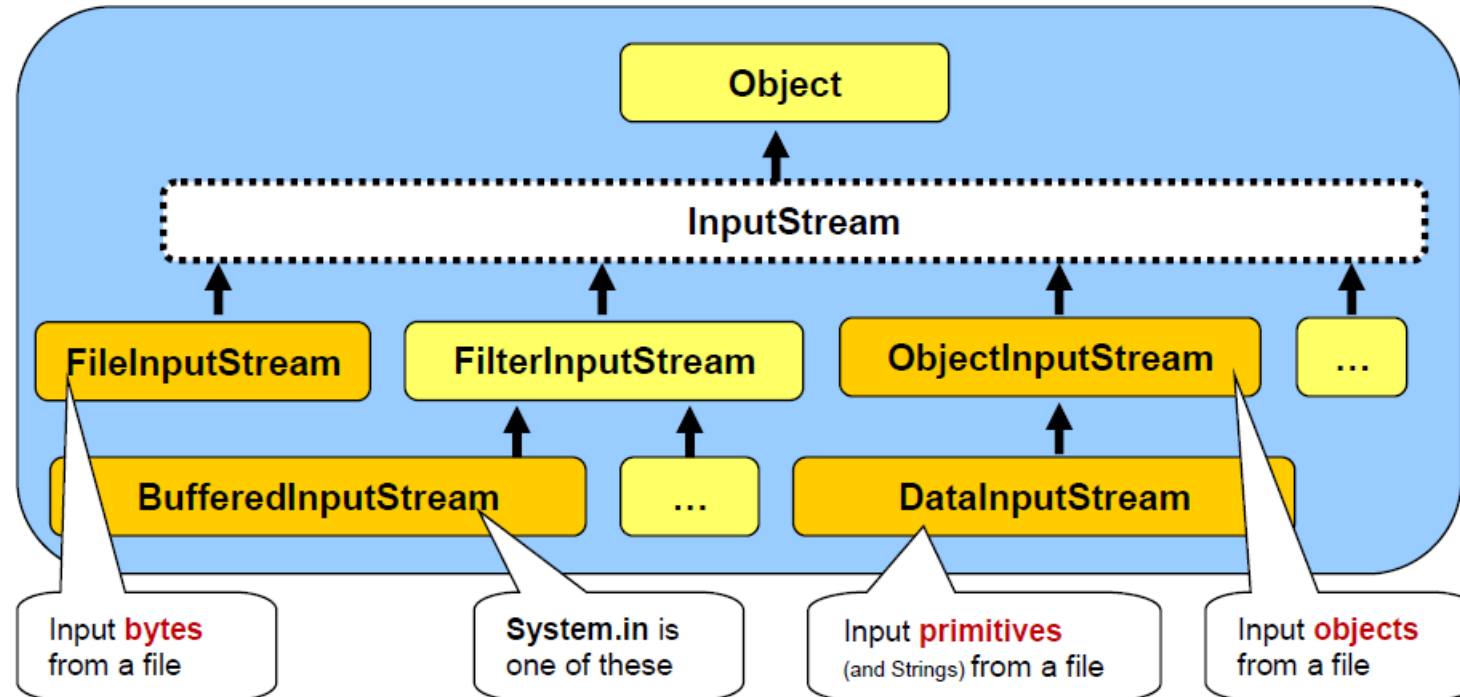
```
writeUTF(String aString)  
writeFloat(float aFloat)  
writeDouble(double aDouble)  
writeBoolean(boolean aBool)  
writeChar(char aChar)
```

```
writeInt(int anInt)  
writeLong(long aLong)  
writeShort(short aShort)  
writeByte(int aByte)
```

- The output from a **DataOutputStream** is not very nice to look at (i.e., it is in binary format).

Reading and Writing Binary Data

- Let us now examine how we could read that information back in from the file with a different program. To start, we need to take a look at the **InputStream** sub-hierarchy as follows ...



Reading and Writing Binary Data

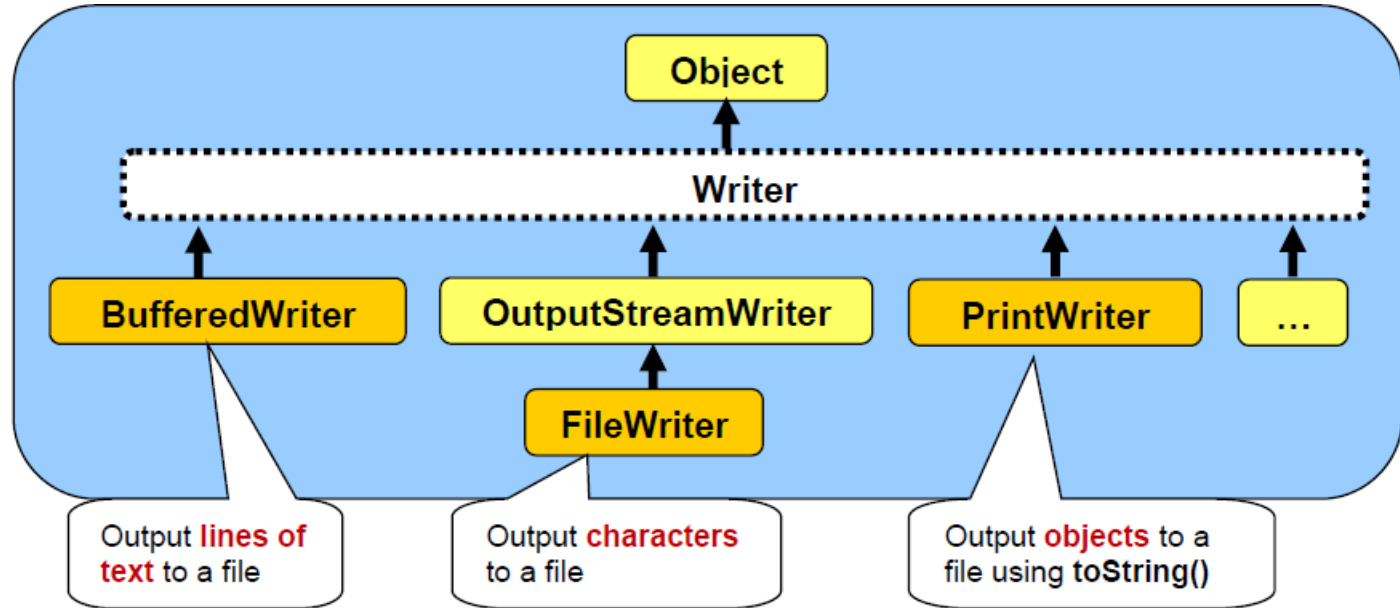
- Notice that it is quite similar to the **OutputStream** hierarchy. In fact, its usage is also very similar. We can read back in the byte data from our file by using **FileInputStream** now as follows:

```
import java.io.*;
public class FileInputStreamTestProgram {
    public static void main(String[] args) {
        try {
            FileInputStream in = new FileInputStream("myFile.dat");
            while (in.available() > 0)
                System.out.print(in.read() + " ");
            in.close();
        } catch (FileNotFoundException e) {
            System.out.println("Error: Cannot open file for reading");
        } catch (IOException e) {
            System.out.println("Error: Cannot read from file");
        }
    }
}
```

- Notice that we now use **read()** to read in a single byte from the file.
- Notice as well that we can use the **available()** method which returns the number of bytes available to be read in from the file (i.e., the file size minus the number of bytes already read in).

Reading and Writing Text Data

- Here is the **Writer** class sub-hierarchy which is used for writing **text** data to a stream:



- Notice that there are 3 main classes that can be used for writing to a text file,
 - Characters**
 - Lines of text**
 - Objects**

Reading and Writing Text Data

- When objects are written to the text file, the **toString()** method for the object is called and the resulting **String** is saved to the file.
- We can output (in text format) to a file using simply the **print()** or **println()** methods with the **PrintWriter** class as follows ...

```
BankAccount aBankAccount;  
PrintWriter out;  
aBankAccount = new BankAccount("Rob Banks");  
aBankAccount.deposit(100);  
out = new PrintWriter(new FileWriter("myAccount2.dat"));  
out.println(aBankAccount.getOwner());  
out.println(aBankAccount.getAccountNumber());  
out.println(aBankAccount.getBalance());  
out.close();
```

- we can actually write any object using the **println()** method. JAVA will use that object's **toString()** method. So if we replaced this code:

```
out.println(aBankAccount.getOwner());  
out.println(aBankAccount.getAccountNumber());  
out.println(aBankAccount.getBalance());
```

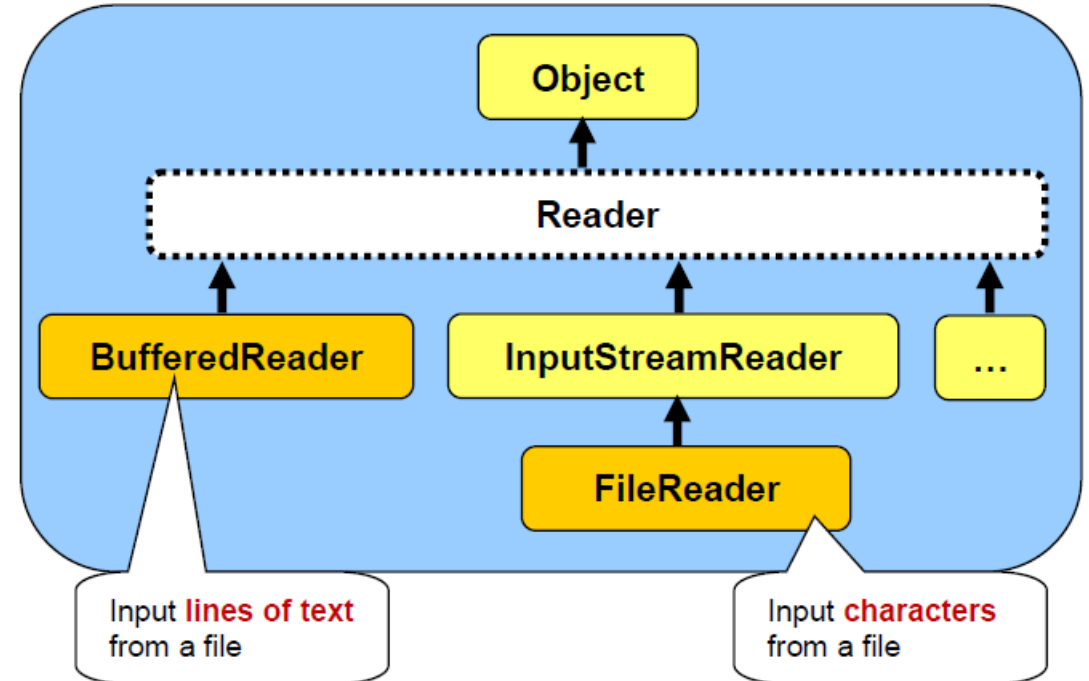
- with this code:

```
out.println(aBankAccount);
```

- we would end up with the following saved to the file:
 - Account #100000 with \$100.0
- So it actually does behave just like the **System.out** console.

Reading and Writing Text Data

- It is also easy to read back in the information that was saved to a text file. Here is the hierarchy of classes for reading text files



- Notice that we can only read-in **characters** and **lines of text** from the text file, but NOT general objects. We will see later how we can re-build read-in objects.
- Most of the time, we will make use of the **BufferedReader** class by using the **readLine()** method as follows

BufferedInputStream/ BufferedOutputStream

- The read()/write() method in InputStream/OutputStream are designed to read/write a single byte of data on each call.
- Overall inefficient, as each call is handled by the underlying operating system (which may trigger a disk access, or other expensive operations).
- **Buffering**, which reads/writes a block of bytes from the external device into/from a memory buffer in a single I/O operation, is commonly applied to speed up the I/O.

Buffering

- Read/Write *blocks of bytes* into **memory buffer**
- Buffer ~ byte array
- Default buffer size ~ 8192bytes

Chained Streams

- Buffering only provides buffering as its core functionality.
- Doesn't take responsibility of dealing with the file on the system.
- Works with or chained with another Stream.



Reading and Writing Text Data

```
BankAccount aBankAccount;  
BufferedReader in;  
in = new BufferedReader(new FileReader("myAccount2.dat"));  
String name = in.readLine();  
int acc = Integer.parseInt(in.readLine());  
float bal = Float.parseFloat(in.readLine());  
aBankAccount = new BankAccount(name, bal, acc); System.out.println(aBankAccount);  
in.close();
```

- Note the use of "primitive data type" wrapper classes to read data types.
- We could have used the **Scanner** class here to simplify the code

```
BankAccount aBankAccount;  
Scanner in = new Scanner(new FileReader("myAccount2.dat"));  
String name = in.nextLine();  
int acc = in.nextInt();  
float bal = in.nextFloat();  
aBankAccount = new BankAccount(name, bal, acc);  
System.out.println(aBankAccount); in.close();
```

Stream IO Operations

1. Open stream
2. Read/ Write
3. Close stream

Standard Template

```
FileInputStream in = null;
try{
    in = new FileInputStream(filename); //open stream
    // read data
} catch (FileNotFoundException ) {
    ...
} finally{
    try{
        if(n != null)
            in.close();
    } catch (IOException e) { ...}
}
```

try – with – resources (Java 7)

java.lang.AutoCloseable



```
try(FileInputStream in = new FileInputStream(filename)) {  
    // Read Data  
} catch (FileNotFoundException e) {  
    ...  
} catch (IOException e) {  
    ...  
}
```

try – with – resources ~ Multiple Resources

```
try(FileInputStream in = new FileInputStream(filename);  
    FileOutputStream out = new FileOutputStream(filename)) {  
    // Read Data  
} catch (FileNotFoundException e) {  
    ... ..  
} catch (IOException e) {  
    ... ..  
}
```

Byte Stream vs Character Stream

Byte Stream	Character Stream
Deal with "Raw Data"	Deal with "Character Data"
Byte by Byte (1 Byte -- 8-bits)	Character by Character (1 char - 16 bits)
Coder responsibility to convert bytes to character	No worries for the coder
Does not always handle Unicode correctly	Handle Unicode appropriately

Which one is more efficient?

Binary I/O does not involve encoding or decoding and is more efficient than text I/O.