CSE201: Monsoon 2022
Advanced Programming

Lecture 13: I/O Streams

Raghava Mutharaju (Section-A)
CSE, IIIT-Delhi
raghava.mutharaju@iiitd.ac.in

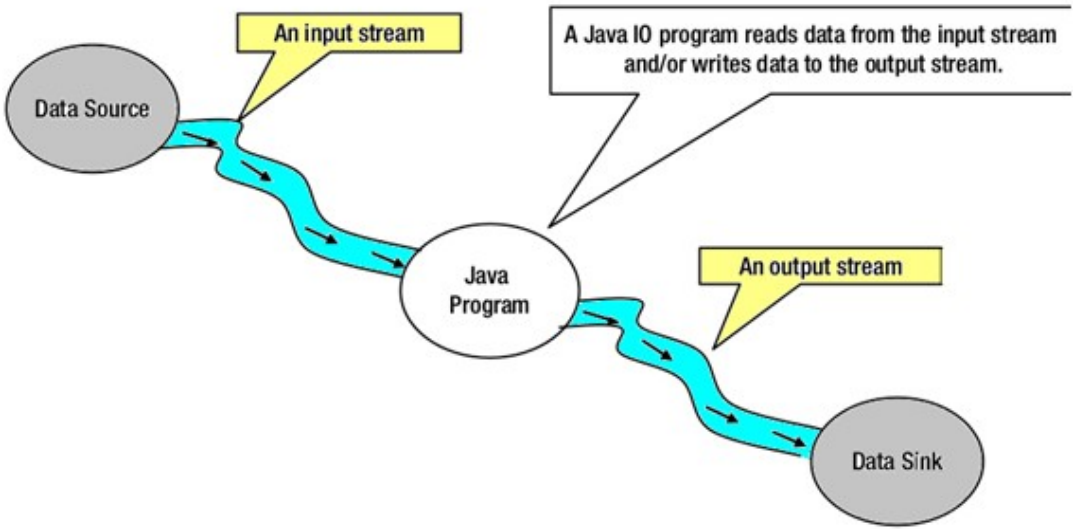
Today’s Lecture

- I/O Streams
- Object serialization and deserialization

Acknowledgements: Oracle Java doc + javatpoint.com

I/O Streams

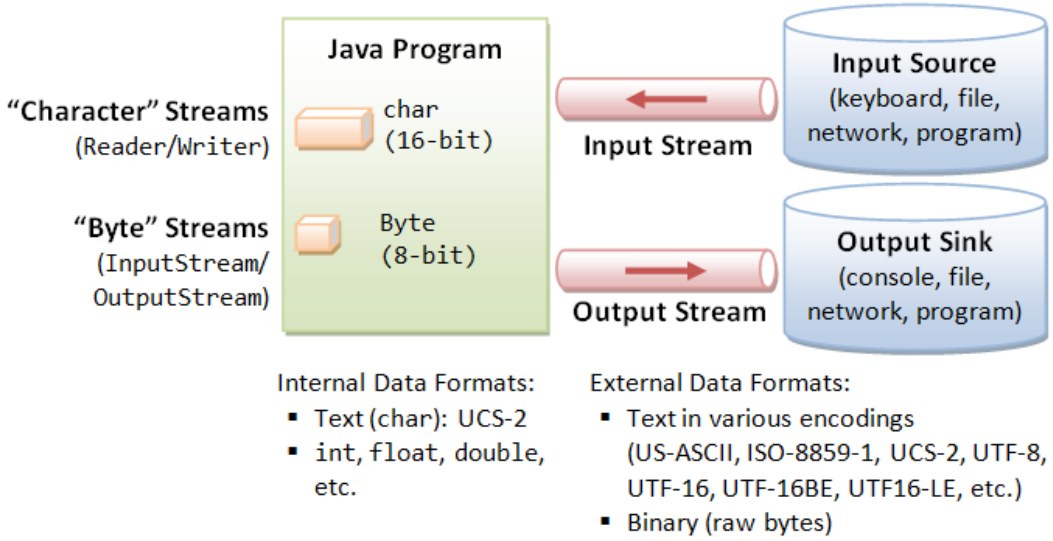
- Stream is a sequence of data
  - Flows in/out the program to/from an external source such as file, network, console, etc.
- Similar to a stream of flowing water…
- Program uses **input stream** to read data from a source, one at a time
- Program uses **output stream** to write data to a destination, one at a time



Streams v/s File Handling

- Stream is a continuous flow of data
  - Streams don’t allow you to move back and forth unlike File
- Streams allows you handle the data the same way irrespective of the location of data (e.g., hard disk, network etc.)
  - You can have the same code to “stream” the data from a file and from the network!

Types of Streams



- Two types of streams
  - Byte stream
  - Character stream
- Byte stream
  - Operates upon stream of "byte" (8-bit)
- Character stream
  - Operates upon stream of "character" Unicode (16-bit)
  - *Unicode* is a computing industry standard designed to consistently and uniquely encode characters used in written languages throughout the world
  - The Unicode standard uses hexadecimal to express a character
    - JVM is platform independent!

`java.io` Package

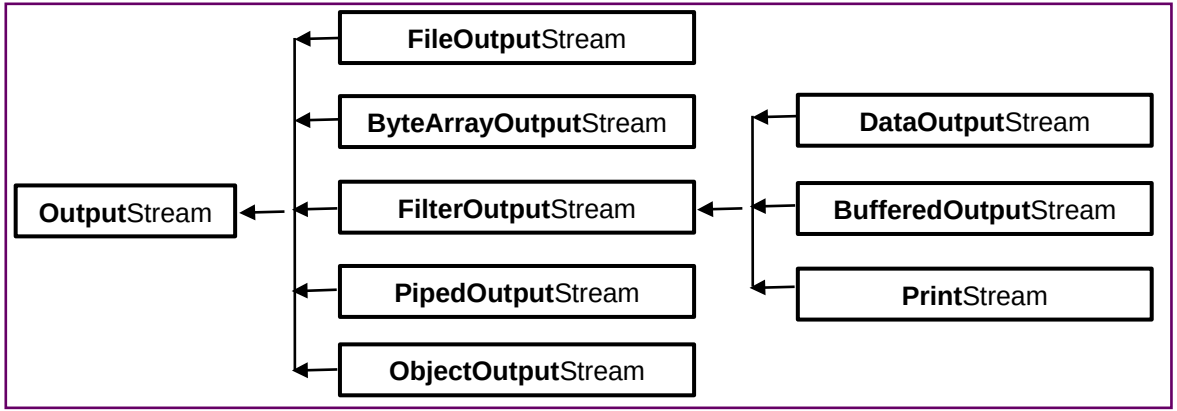
- Reading

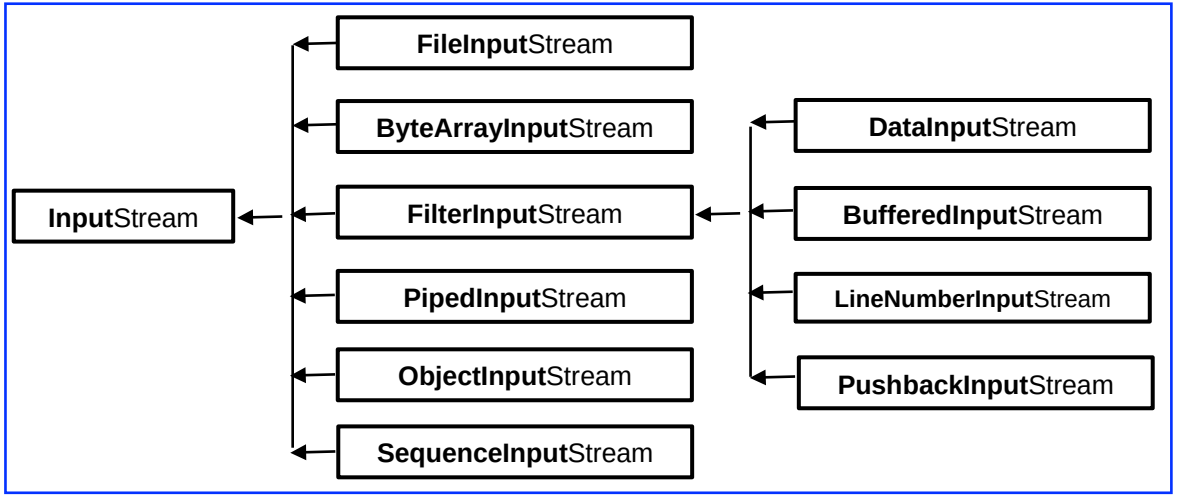
  open a stream
  while more information
      read information
  close the stream

- Writing

  open a stream
  while more information
      write information
  close the stream

Byte Stream Hierarchy





- OutputStream
  - This is the abstract class
  - Parent class of all classes representing an output stream of bytes
  - An output stream accepts output bytes and sends them to some sink
- InputStream
  - This is the abstract class
  - Parent class of all classes representing an input stream of bytes

Byte Streams in System Class

```
public final class System {

    public static final InputStream in;

    public static final PrintStream out;

    public static final PrintStream err;

    ...........

}


public static void main(String args[]) {
    Scanner in = new Scanner(System.in);
//java.lang
    // Scanner class implements iterator
    while (in.hasNext()) {
        System.out.println(in.next());
    }
    in.close();
}
```

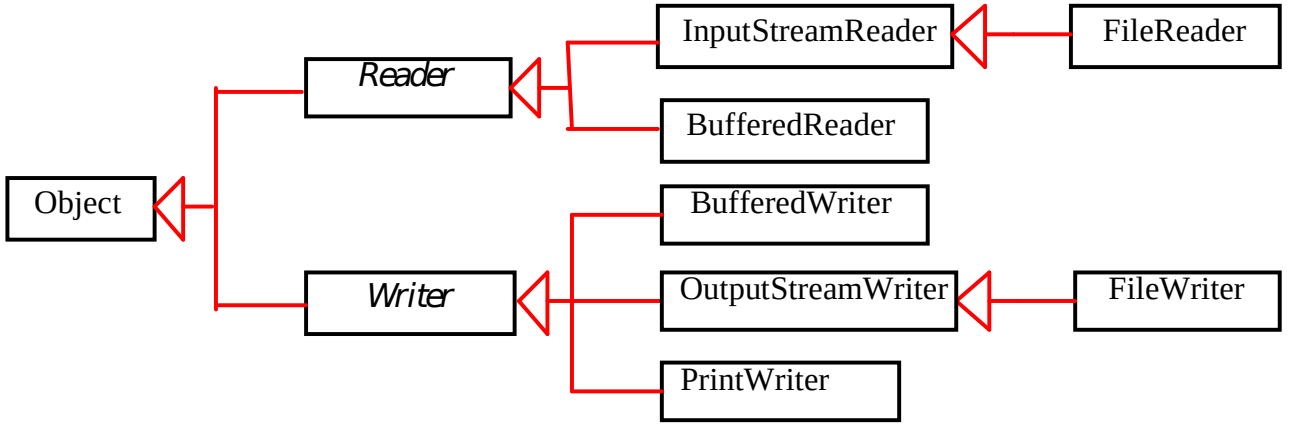
- In java, 3 streams are created for us automatically. All these streams are attached with console
  - **System.out:** standard output stream
  - **System.in:** standard input stream
  - **System.err:** standard error stream

Byte Stream Example

```
public static void main(String args[])
            throws IOException
{
    FileInputStream in = null;
    FileOutputStream out = null;
    try {
        // both constr. throws FileNotFoundException
        in = new FileInputStream("input.txt");
        out = new FileOutputStream("output.txt");
        int c;
        while ((c = in.read()) != -1) { //
IOException
            out.write(c);       // IOException
        }
    } finally {
        if (in != null)
            in.close();         // IOException
        if (out != null)
            out.close();        // IOException
        }
}
```

- InputStream
  - **`read()`** – read the next byte of data from the input stream
  - **`close()`** – close input stream
- OutputStream
  - **`write(int)`** – write a byte to current output stream
  - **`close()`** – close output stream
- Byte stream is used for low-level I/O, e.g., processing binary files

Character Stream Hierarchy



- All character stream classes are subclasses of `Reader` and `Writer` class
- Used for processing text files (character by character)

Character Stream Example

```
public static void main(String args[])
            throws IOException
{
    FileReader in = null;
    FileWriter out = null;
    try {
        // both constr. throws FileNotFoundException
        in = new FileReader("input.txt");
        // throws IOException
        out = new FileWriter("output.txt");
        int c;
        while ((c = in.read()) != -1) { //
IOException
            out.write(c);       // IOException
        }
    }finally {
        if (in != null)
            in.close();              // IOException
        if (out != null)
            out.close();        // IOException
        }
}
```

- This example is very similar to the byte stream I/O
- In terms of coding, the difference is in using FileReader and FileWriter for input and output
- Note that “int” type variable is used in both these examples to read and write. Although internally they are working differently:
  - In byte stream example, the “int” variable holds a byte value in last 8 bits
  - In this example, the “int” variable holds character value in its last 16 bits

Buffered Streams (1/2)

```
public static void main(String args[])
            throws IOException
{
    BufferedReader in = null;
    PrintWriter out = null;
    try {
        in = new BufferedReader( new
                FileReader("input.txt"));
        out = new PrintWriter( new
                FileWriter("output.txt"));
        String l;
        while ((l = in.readLine()) != null){
//IOException
            out.println(l); // does not throw IOException
        }
    }finally {
        if (in != null)
            in.close();                 // IOException
        if (out != null)
            out.close();          // IOException
        }
}
```

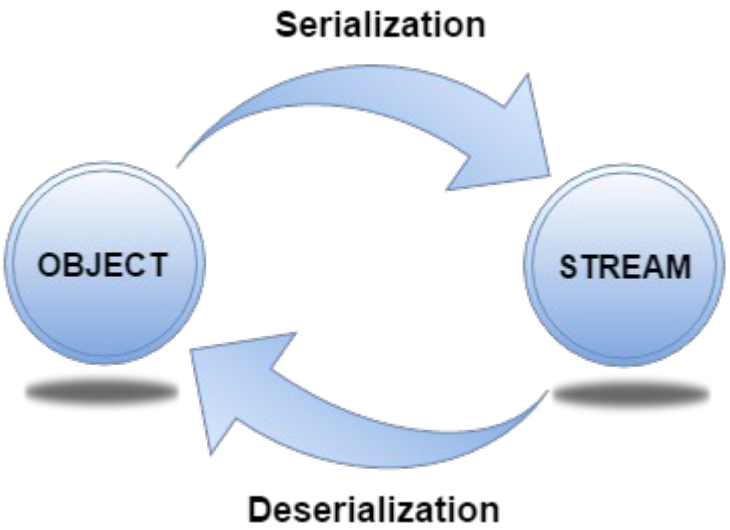
- Combine streams into chains to achieve more advanced input and output operations
- Reading character by character from a file is slow
- Faster to read a larger block of data from the disk and then iterate through that block byte by byte afterwards
- The code on the left does input and output one line at a time
  - Unlike **BufferedWriter**, **PrintWriter** swallows exceptions and provide methods such as println(), etc.

Buffered Streams (2/2)

```
public static void main(String args[])
          throws IOException
{
    Scanner in = null;
    PrintWriter out = null;
    try {
        in = new Scanner( new BufferedReader( new
               FileReader("input.txt")));
        out = new PrintWriter( new
               FileWriter("output.txt"));
        while (in.hasNext()) {
            out.println(in.next());
        }
    }finally {
        if (in != null)
            in.close();
        if (out != null)
            out.close();
        }
}
```

- Here we are combining three classes for breaking input into tokens:
  - Scanner
  - BufferedReader
  - FileReader
- BufferedReader will read one line at a time and Scanner will be able to parse this line by white space separated tokens

Serialization and Deserialization



- Serialization in Java is a mechanism of writing the state of an **object** into a **byte stream**
  - **Note:** it's the object state that is recorded but not the actual class definition ("class file")
- The reverse operation is called deserialization
- Some usage
  - Storing live objects in a file
  - Hibernating applications
  - Moving object state over the network (marshaling)

java.io.Serializable Interface

```
package java.io;
public interface Serializable { /*empty*/ }
```

- Must be implemented by the class to be serialized
- This is a tag/marker interface similar to Cloneable interface
  - Hint to JVM !

Example: Serializing and Deserializing

```
1. import java.io.*;
2.
3. class Manager implements Serializable {
4.     private String name;
5.     public Manager(String n) { ..... }
6. }

7. public class Client {
8.    public static void serialize()
9.                          throws IOException {
10.        Manager s1 = new Manager("Amy");
11.        ObjectOutputStream out = null;
12.        try {
13.           out = new ObjectOutputStream (
14.                 new
FileOutputStream("out.txt"));
15.           out.writeObject(s1);
16.        } finally {
17.           out.close();
18.        }
19.   }
20.
/* Continued on RHS window */
```

```
    /* Continued from LHS window */
21.    public static void deserialize()
22.       throws IOException, ClassNotFoundException {
23.        ObjectInputStream in = null;
24.        try {
25.            in =  new ObjectInputStream (
26.                    new FileInputStream("out.txt"));
27.            Manager s1 = (Manager) in.readObject();

29.        } finally {
30.            in.close();
31.        }
32.    }
33.
34.    public static void main(String[] args)
35.        throws IOException,ClassNotFoundException {
36.
37.       serialize();
39.       deserialize();
40.    }
41. } /* End of Main class */
```

Suppose you have a Client.java that only has the above deserializes() method. Compilation of Client.java will generate two class files Client.class and Manager.class. If you try running "java Client" without Manager.class in its classpath then ClassNotFoundException will be thrown at Line 27 above.

Rules for Serializing (1/3)

```
import java.io.*;
class Address {
    private String city;
    public Address(String c) { ..... }
}
class Manager implements Serializable {
     private String name;
     private Address addr;
     public Manager(String n, String city) { ..... }
}
public class Main {
  public static void serialize() throws IOException {
        Manager s1 = new Manager("Amy", "Delhi");
        ObjectOutputStream out = null;
        try {
           out = new ObjectOutputStream (
                 new FileOutputStream("out.txt"));
           out.writeObject(s1);
        } finally {
           out.close();
        }
    }
    ......
}
```

- This program compiles fine but will generate NotSerializableException
- All fields of Manager class should either be primitive type or serializable objects
  - Address class should also implement Serializable interface

Rules for Serializing (2/3)

```
import java.io.*;
class Employee {
    private String address;
    public Employee(String a) { ..... }
}
class Manager extends Employee
                        implements Serializable {
    private String name;
    public Manager(String n, String city) { ..... }
}
public class Main {
  public static void serialize() throws IOException
{
        Manager s1 = new Manager("Amy", "Delhi");
        ObjectOutputStream out = null;
        try {
           out = new ObjectOutputStream (
                 new FileOutputStream("out.txt"));
           out.writeObject(s1);
        } finally {
           out.close();
        }
    }
    ......
}
```

- This program compiles fine but will generate **`InvalidClassException`** while typecasting the object to Manager type during deserialization
- Two ways to fix this issue:
  1. Provide a default constructor in Employee
  2. Or, implement Serializable in Employee class (superclass)
     - This is obviously the safer and easier choice
     - With this change, its not required in the Manager (or any subclass) to mention implements Serializable

Rules for Serializing (3/3)

- There is no point in serializing static field members in a class
  - Static fields do not represents object state but they represent class state
    - Recall, static variables are accessed using class name and not with objects of the class
  - There will not be any compilation/runtime issue, although the value serialized will not make any sense as it can always be updated later in the class
- `transient` keyword in Java
  - If you don't want any field to be serialized then mark that as "`transient`"

`serialVersionUID` in Serialization (1/2)

- What happens when we compile the programs in previous slides by enabling warning?

$ javac –Xlint Main.java

Main.java:3: warning: [serial] serializable class Manager has no definition of serialVersionUID

`serialVersionUID` in Serialization (2/2)

- JVM generates a unique `serialVersionUID` to each class implementing Serializable interface

```
ANY-ACCESS-MODIFIER static final long serialVersionUID = 42L;
```

- Its always advisable to declare this variable in each serializable class with your own number of choice
- Helpful in verifying if the object being deserialized is of the same type of the specified class
  - The class declaration might have got updated (e.g. added new fields) after serialization and now deserializing the object will generate `InvalidClassException`

Where are we as of now

- **CSE201 Post Conditions**
  1. Students are able to demonstrate the knowledge of basic principles of Object Oriented Programming such as encapsulation (classes and objects), interfaces, polymorphism and inheritance; by implementing programs ranging over few hundreds lines of code
  2. Implement basic event driven programming, exception handling, and threading
     - Already covered little bit of event driven programming in refresher module (Day 3) but we will see more
  3. Students are able to analyze the problem in terms of use cases and create object oriented design for it. Students are able to present the design in UML
     - Already covered little bit of UML but we will see more
  4. Students are able to select and use a few key design pattern to solve a given problem in hand
  5. Students are able to use common tools for testing (e.g., JUnit), debugging, and source code control as an integral part of program development
     - Will turn green by end of this week

Next Lecture

- Unit testing using JUnit
- Inner classes