CSE201: Monsoon 2024
Advanced Programming

Lecture 14: I/O Streams

Dr. Arun Balaji Buduru

Head, Center of Technology in Policing

Founding Head, Usable Security Group (USG)

Associate Professor, Dept. of CSE | HCD

IIIT-Delhi, India

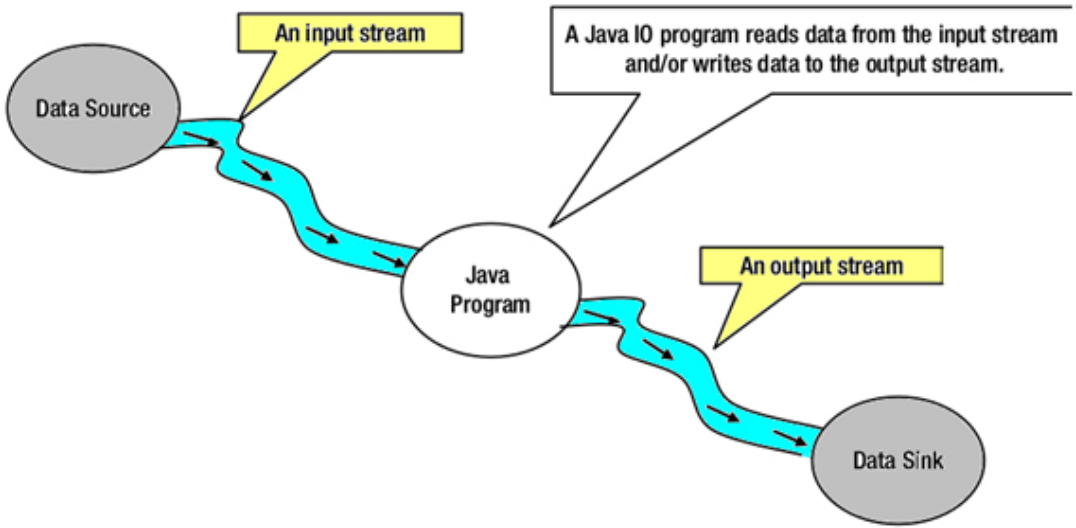
Today’s Lecture

- I/O Streams
- Object serialization and deserialization

Acknowledgements: Oracle Java doc + javatpoint.com

I/O Streams

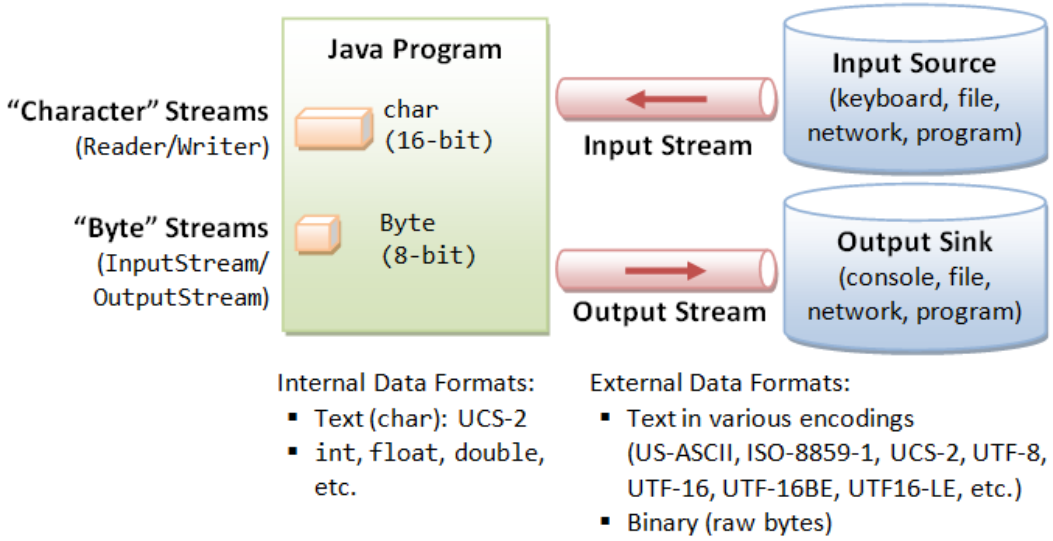
- Stream is a sequence of data
  - Flows in/out the program to/from an external source such as file, network, console, etc.
- Similar to a stream of flowing water…
- Program uses **input stream** to read data from a source, one at a time
- Program uses **output stream** to write data to a destination,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

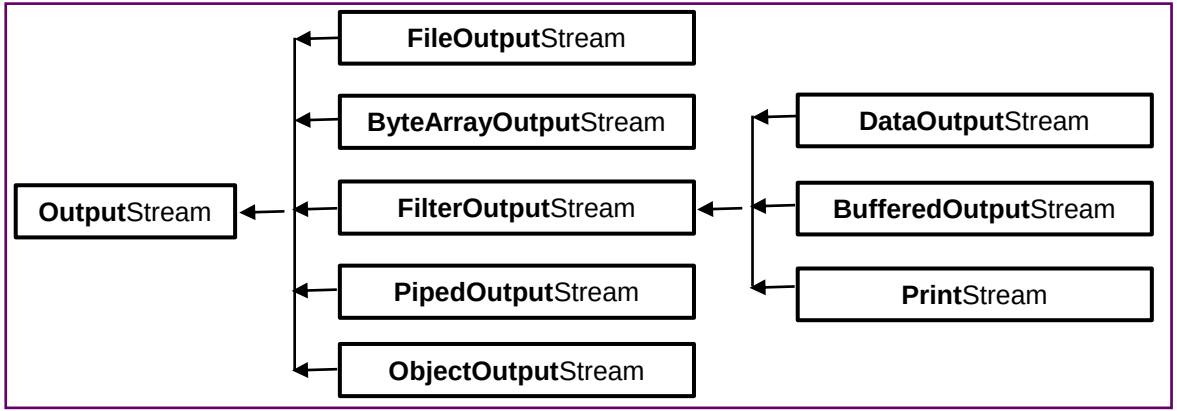
  open a stream

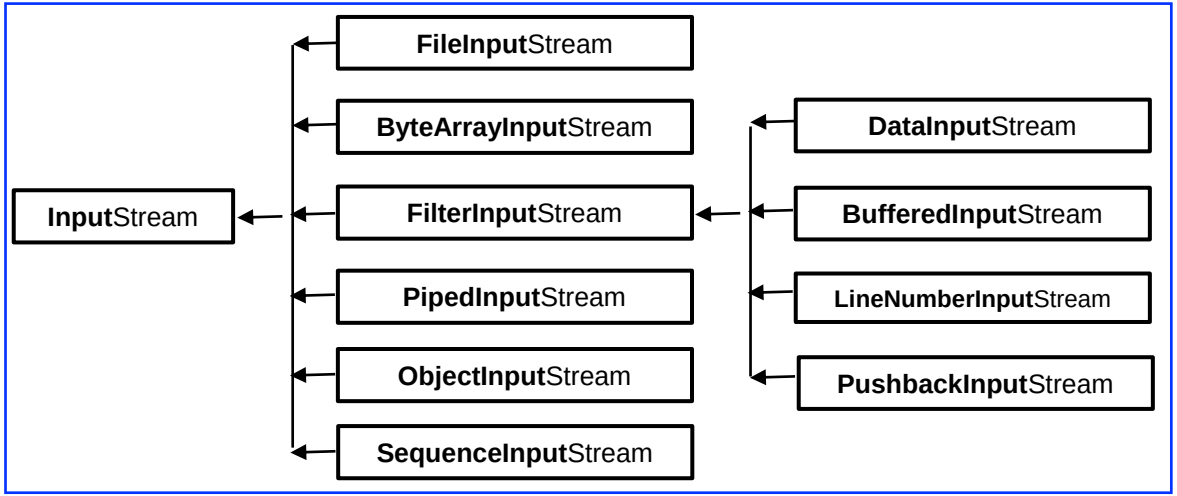
  while more information

  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();
}
```

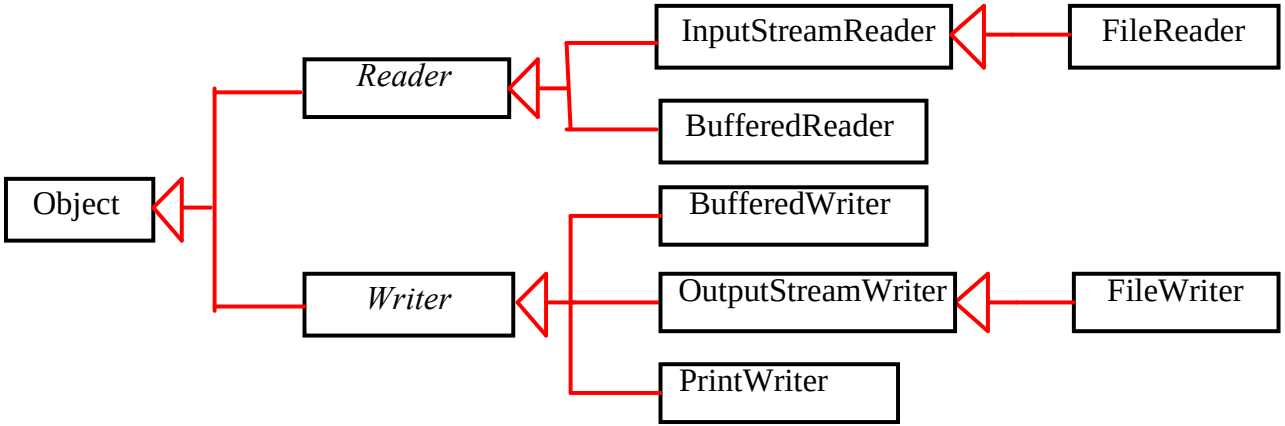
- In java, 3 streams are created for us automatically. All these streams are attached with console
  - **System.out:** standard output stream
  - **System.in:** standard input stream
  - **System.err:** standard error stream

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

Data Stream

```
public static void main(String[] args)
                                throws IOException
{
    int[] empid = {1, 2, 3};
    String[] name = {"John", "Joe", "Amy"};
    DataOutputStream out = null;
    try {
        out = new DataOutputStream(new
                BufferedOutputStream(new
                FileOutputStream("output.txt")));
        for (int i = 0; i < empid.length; i ++) {
           out.writeInt(empid[i]);
           out.writeUTF(name[i]);
        }
    } catch(EOFException e) {
        // do nothing
    }
    finally {
        out.close();
    }
}
```

- Supports binary I/O of primitive data type values
  - The resulting output is not human-readable but reading it back in will be faster than parsing text
  - DataOutputStream implements the DataOutput interface
- To read the data back in, there are methods defined in the DataInput interface
  - DataInputStream implements the DataInput interface
- DataStreams detects an end-of-file condition by catching EOFException, instead of testing for an invalid return

Directories in Java

```
public static void main(String[] args)  {

    String dirname = “/tmp/vivek”; // works on Windows too
    File f = new File(dirname);

    // creating the directory
    f.mkdirs();

    String[] paths = (new File(“/tmp”)).list();

    // List all the files and directory under “/tmp”
    for(String path: paths) {
        System.out.println(path);
    }
}
```

- **File** object can be used to create directories and to list down files available in a directory
  - mkdir( ) – creates single directory
  - mkdirs( ) – create entire directory structure
  - list( ) – lists down all the files and directory under a given directory