Java.io package

Fresher Learning Program
January, 2012

People matter, results count.

Objectives of java.io package

- Purpose:
  - To understand various classes of IO package and how to use them for date input and output
- Product:
  - To know what is Stream and stream class in Java
  - To know about Stream class hierarchy and various types of streams
  - File system
- Process:
  - Theory Sessions along with assignments
  - A recap at the end of the session in the form of Quiz

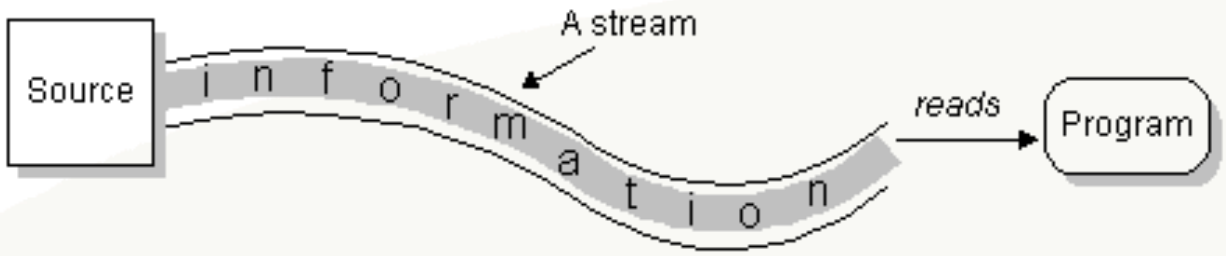
Table of Contents

- Stream
- Stream class hierarchy
- Control flow of an I/O operation using Streams
- Different types of stream
  - Byte streams
  - Character streams
  - Buffered streams
  - Standard I/O streams
  - Data streams
  - Object streams
- File class

Streams

- What is a stream?

- Stream represents a flow of data in one direction. A program opens a *stream* on an information source (a file, memory, a socket) and reads the information sequentially, as shown here:



- Similarly, a program can send information to an external destination by opening a stream to a destination and writing the information out sequentially.

I/O Streams

- An I/O Stream represents an input source or an output destination

- A stream can represent many different kinds of sources and destinations

  – disk files, devices, other programs, a network socket, and memory arrays

- Streams support many different kinds of data

  – simple bytes, primitive data types, localized characters, and objects

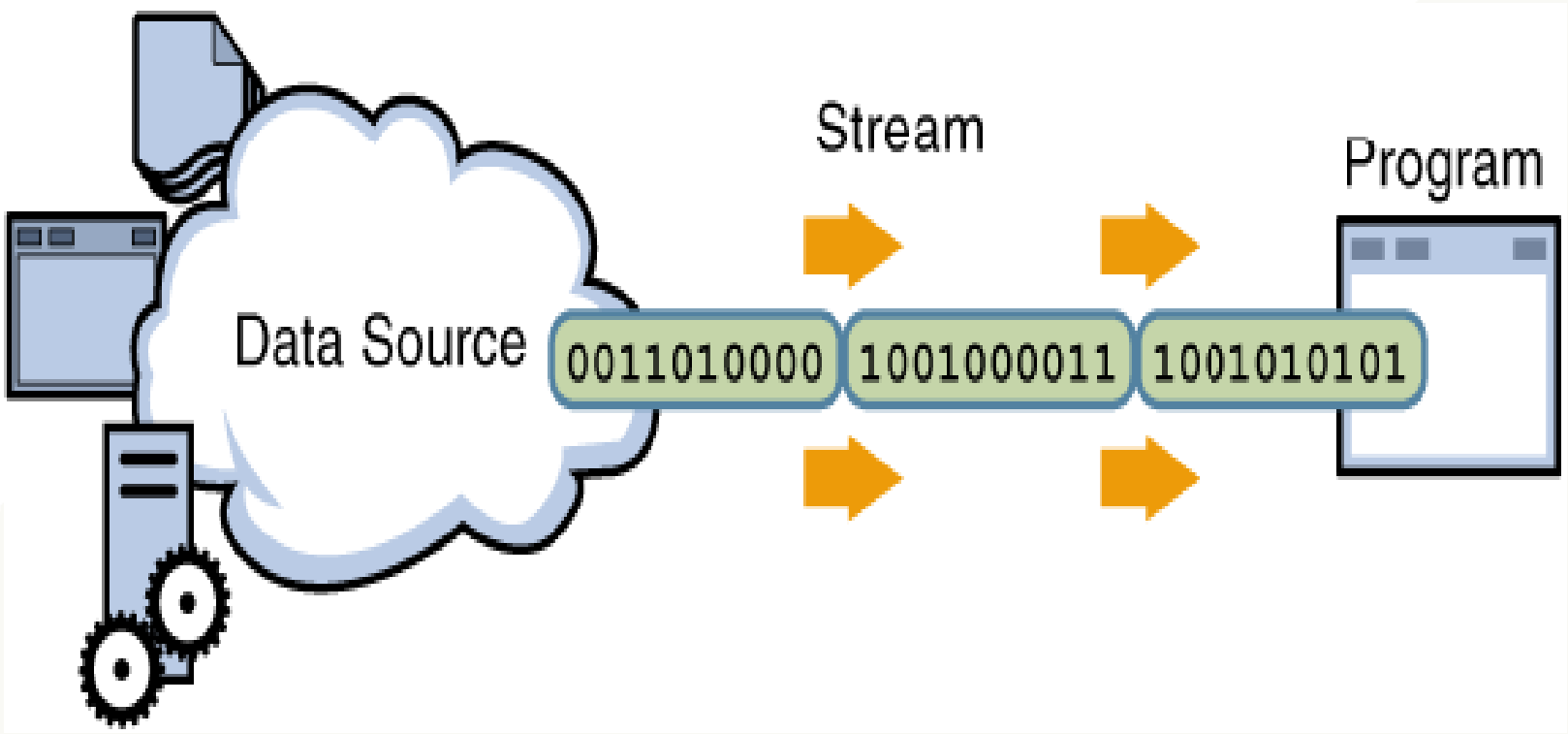
I/O Streams

- Some streams simply pass on data; others manipulate and transform the data in useful ways.

- No matter how they work internally, all streams present the same simple model to programs that use them
  – A stream is a sequence of data

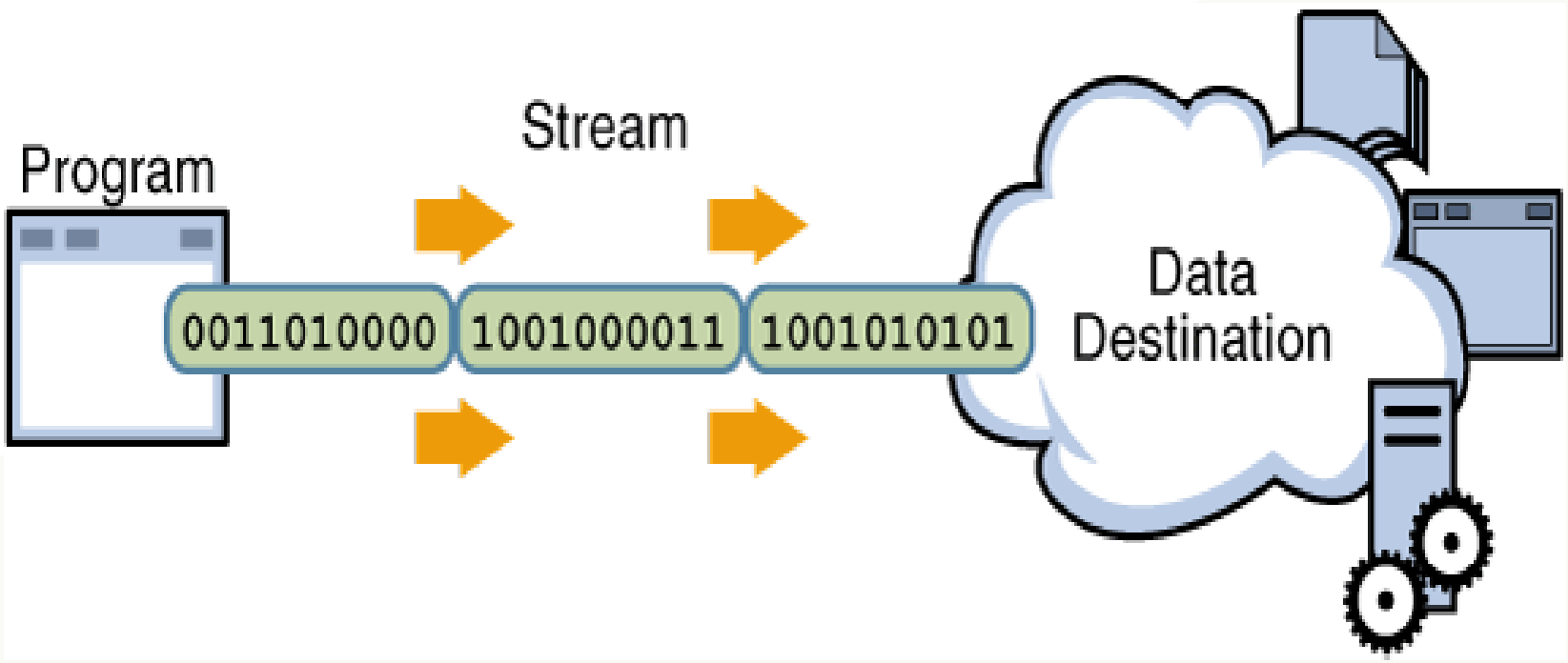
Input Stream

- A program uses an input stream to read data from a source, one item at a time



Output Stream

- A program uses an output stream to write data to a destination, one item at time



Input and Output Streams

Input or source streams can read from these streams

– Root classes of all input streams:

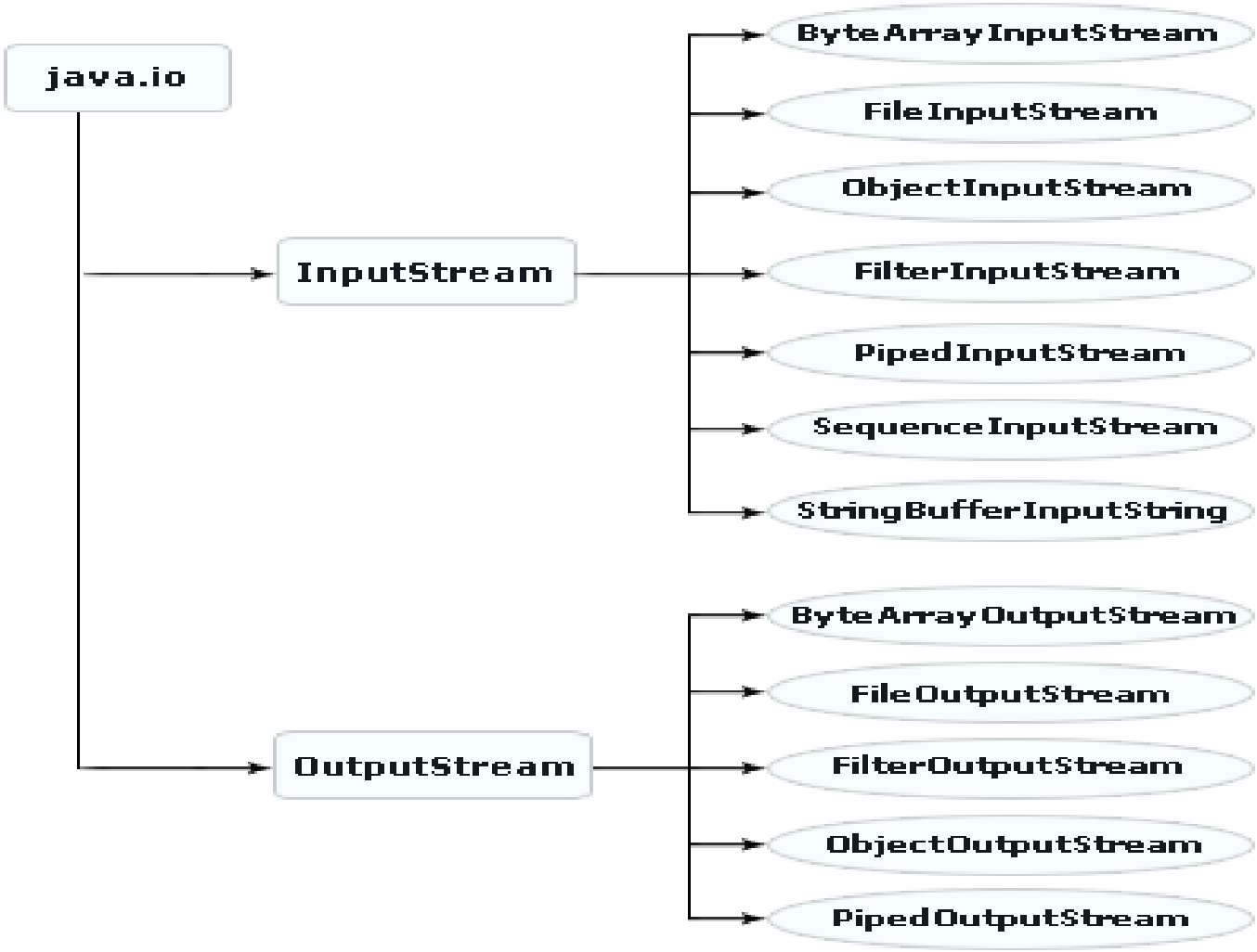
- The InputStream Class
- The Reader Class

Output or sink (destination) streams can write to these streams

– Root classes of all output streams:

- The OutputStream Class
- The Writer Class

Stream Class Hierarchy



Control Flow of I/O Operation Using Streams

Streams - Classes

Java Stream Classes are basically categorized as

- ByteStream
  - InputStream
  - OutputStream
- CharacterStream
  - Reader
  - Writer

Character and Byte Streams

Byte streams

- For binary data
- Root classes for byte streams:
- The *InputStream Class*
- The *OutputStream Class*
- Both classes are *abstract*
-

Character streams

- For Unicode characters
- Root classes for character streams:
- The *Reader class*
- The *Writer class*
- Both classes are *abstract*

Control Flow of an I/O operation

- Create a stream object and associate it with a data source (data-destination)
- Give the stream object the desired functionality through stream chaining
- while (there is more information) read(write) next data from(to) the stream close the stream

Byte Stream

- Programs use byte streams to perform input and output of 8-bit bytes
- All byte stream classes are descended from *InputStream and OutputStream*
- There are many byte stream classes FileInputStream and FileOutputStream
- They are used in much the same way; they differ mainly in the way they are constructed

FileInputStream & FileOutputStream

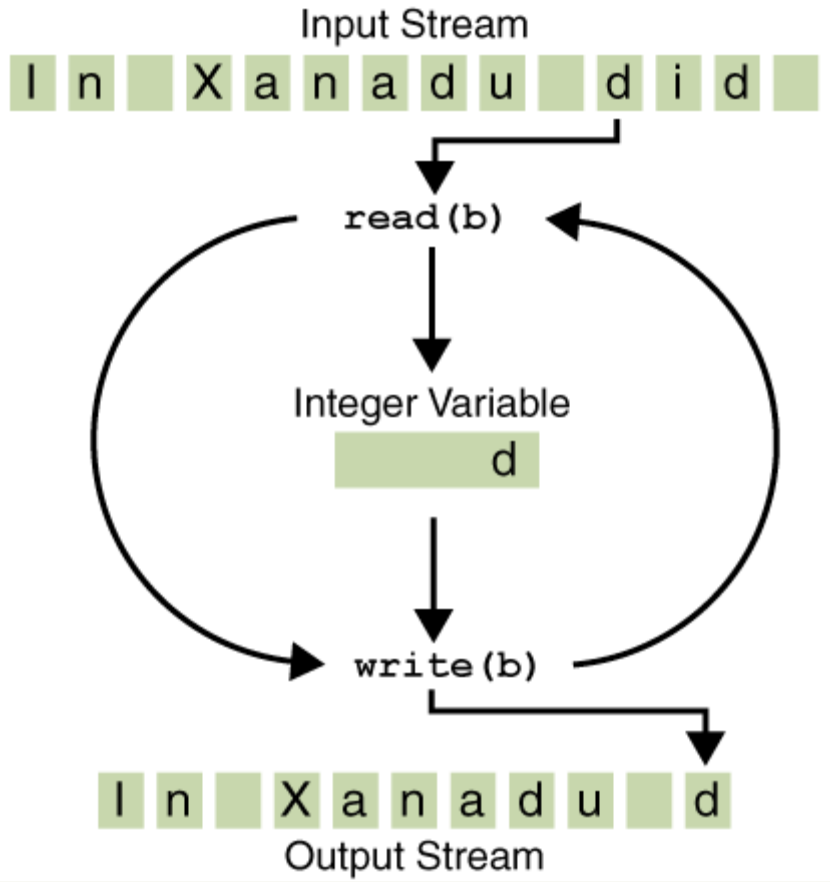
Example:

```
public class CopyBytes {
  public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
              in = new FileInputStream("xanadu.txt");
              out = new FileOutputStream("outagain.txt");
              int c;
              while ((c = in.read()) != -1) {
                    out.write(c);
              }
        }
```

FileInputStream & FileOutputStream

```
finally {
                    if (in != null) {
                            in.close();
                    }
                    if (out != null) {
                            out.close();
                    }
            }
    }
}
```

Simple Byte Stream input and output



Character Stream

- The Java platform stores character values using Unicode conventions
- Character stream I/O automatically translates this internal format to and from the local character set.
  - In Western locales, the local character set is usually an 8-bit superset of ASCII.
- All character stream classes are descended from Reader and Writer
- As with byte streams, there are character stream classes that specialize in file I/O: FileReader and FileWriter.

FileReader & FileWriter

Example:

```
public class CopyCharacters {
        public static void main(String[] args) throws IOException {
                FileReader inputStream = null;
                FileWriter outputStream = null;
                try {
                    inputStream = new FileReader("xanadu.txt");
                    outputStream = new FileWriter("characteroutput.txt");
                    int c;
                    while ((c = inputStream.read()) != -1) {
                            outputStream.write(c);
                    }
                }
```

FileReader & FileWriter...

```
    finally {
            if (inputStream != null) {
                    inputStream.close();
            }
            if (outputStream != null) {
                    outputStream.close();
            }
        }
    }
}
```

Buffered Streams

- **Why Buffered Streams?**

- An unbuffered I/O means each read or write request is handled directly by the underlying OS

  – This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive.

- To reduce this kind of overhead, the Java platform implements buffered I/O streams

  – Buffered input streams read data from a memory area known as a buffer; the native input API is called only when the buffer is empty

Buffered Streams

– Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.

- Example

```
inputStream =
new BufferedReader(new FileReader("xanadu.txt"));
outputStream =
new BufferedWriter(new FileWriter("characteroutput.txt"));
```

Buffered Streams…

- **Buffered Stream Classes**

- *BufferedInputStream and BufferedOutputStream* create buffered byte streams

- *BufferedReader and BufferedWriter create buffered* character streams

Data Streams

- Data streams support binary I/O of primitive data type values (boolean, char, byte, short, int, long, float, and double) as well as String values

- All data streams implement either the *DataInput* interface or the *DataOutput interface*

- DataInputStream and DataOutputStream are most widely-used implementations of these interfaces

- *DataOutputStream can only be created as a wrapper* for an existing byte stream object

```
out = new DataOutputStream(
new BufferedOutputStream(
new FileOutputStream(dataFile)));
```

DataInputStream

- Like *DataOutputStream, DataInputStream must be* constructed as a wrapper for a byte stream

- End-of-file condition is detected by catching *EOFException, instead of testing for an invalid return* value

```
in = new DataInputStream(
new BufferedInputStream(
new FileInputStream(dataFile)));
try{
   double price = in.readDouble();
   int unit = in.readInt();
   String desc = in.readUTF();
} catch (EOFException e){
}
```

Object Streams

- Object streams support I/O of objects
  - Like Data streams support I/O of primitive data types
  - The object has to be *Serializable type*

- The object stream classes are ObjectInputStream and ObjectOutputStream
  - These classes implement ObjectInput and ObjectOutput, which are subinterfaces of DataInput and DataOutput
  - An object stream can contain a mixture of primitive and object values

Input and Output of Complex Object

- The writeObject and readObject methods are simple to use, but they contain some very sophisticated object management logic

  – This isn't important for a class like Calendar, which just encapsulates primitive values. But many objects contain references to other objects.

- If readObject is to reconstitute an object from a stream, it has to be able to reconstitute all of the objects the original object referred to.

– These additional objects might have their own references, and so on.

WriteObject

- The writeObject traverses the entire web of object references and writes all objects in that web onto the stream

- A single invocation of writeObject can cause a large number of objects to be written to the stream.

Always Close Streams

- Closing a stream when it's no longer needed is very important — so important that your program should use a finally block to guarantee that both streams will be closed even if an error occurs

- This practice helps avoid serious resource leaks.

File Class

- Not a stream class
- Important since stream classes manipulate *File* objects
- Abstract representation of actual files and directory pathname
- Has four constructors

Example to copy a File

```
import java.io.*;
public class CopyBytes {

        public static void main(String[] args)  {
                File inputFile = new File(args[0]);
                File outputFile = new File(args[1]);

                try {
                        FileInputStream in = new FileInputStream(inputFile);
                        FileOutputStream out = new FileOutputStream(outputFile);
                        int c;
                        while ((c = in.read()) != -1)
                                out.write(c);
                        in.close();
                        out.close();
                } catch(FileNotFoundException fnfe)  {
                        // do something
                } catch (IOException ioe) {
                        // do something }
                }
}
```

Recap (important keywords)

Input stream

output stream

Data stream

write object

file class

character and byte

Thank You For Your Time

People matter, results count.