### Java I/O Stream

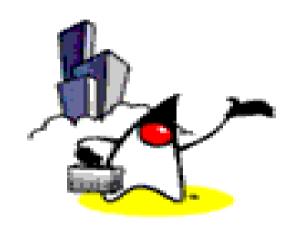
Sang Shin
Michèle Garoche
www.javapassion.com
"Learn with Passion!"



### **Topics**

- What is an I/O stream?
- Types of Streams
- Stream class hierarchy
- Stream chaining
- Byte streams
- Character streams
- Buffered streams
- Standard I/O streams
- Data streams
- Object streams
- File class





# What is an I/O Stream?

### **I/O Streams**

- A stream is a sequence of data
- A stream is associated with different kinds of sources and destinations
  - disk files, devices, other programs, a network socket, and memory arrays
- Streams support different kinds of data
  - simple bytes, localized characters, primitive data types, and objects
- Some streams simply pass on data; others manipulate and transform the data in useful ways.



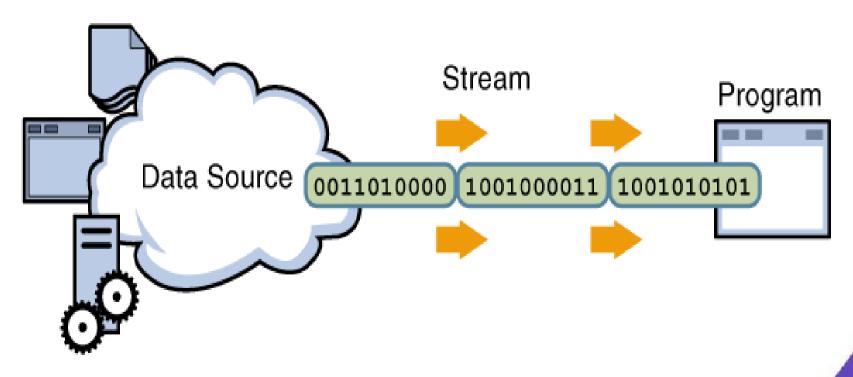
### **Programming Model**

 No matter how they work internally, all streams work in consistent and simple programming model



### **Input Stream**

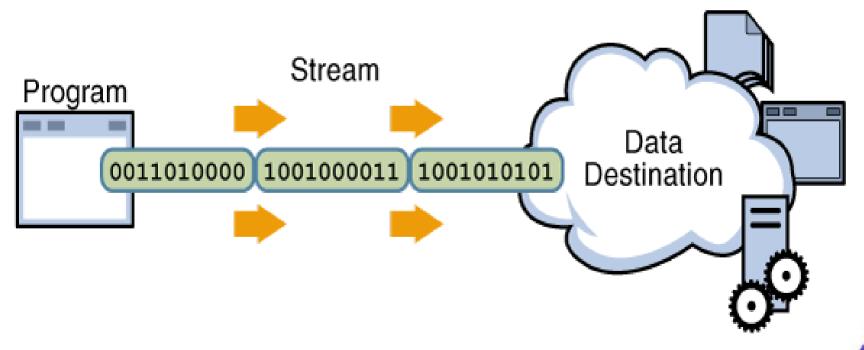
 A program uses an input stream to read data from a source, one item at a time



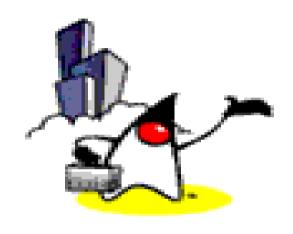


### **Output Stream**

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







## Types of Streams

### **Types of Streams**

- Character and Byte Streams
  - Character vs. Byte
- Input and Output Streams
  - Based on source or destination
- Node and Filter Streams
  - Whether the data on a stream is manipulated or transformed or not



### **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



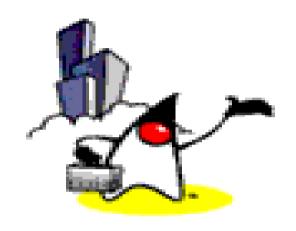
### **Input and Output Streams**

- 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

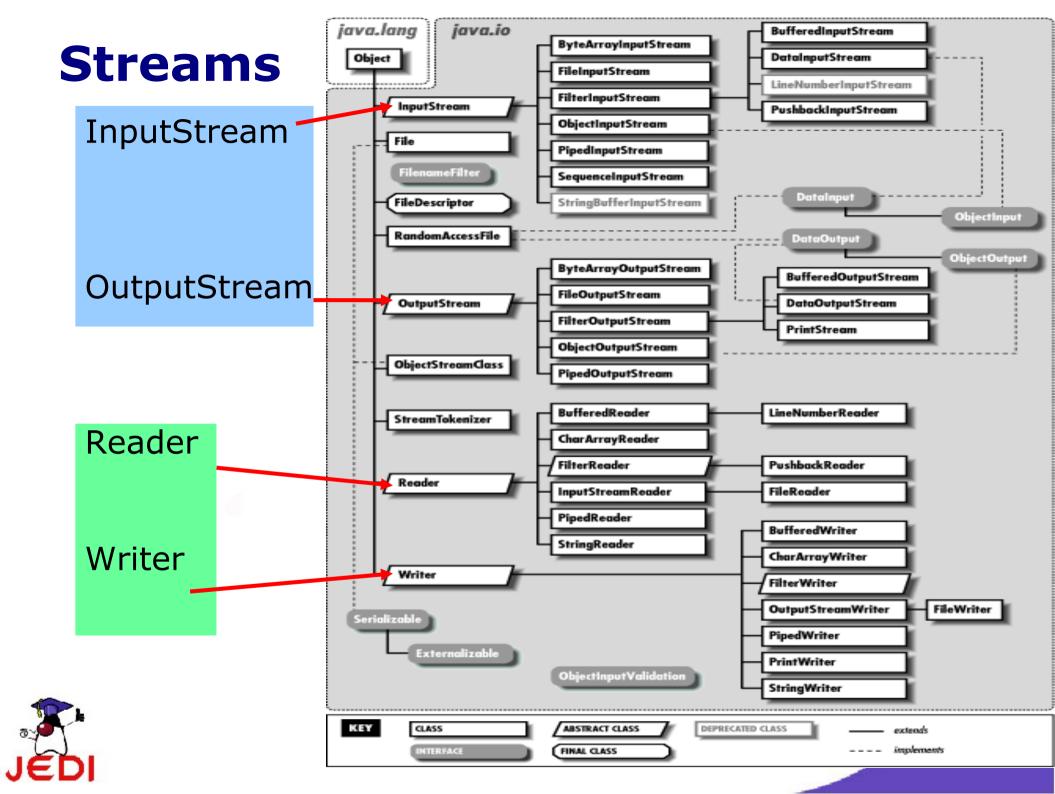


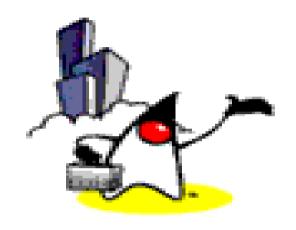
#### **Node and Filter Streams**

- Node streams (Data sink stream)
  - Contain the basic functionality of reading or writing from a specific data source
  - Types of node streams include files, memory and pipes
- Filter streams (Processing stream)
  - Layered onto node streams
  - For additional functionality- altering or managing data in the stream
- Adding layers to a node stream is called stream chaining



# Stream Class Hierarchy





## Stream Chaining

### Stream Chaining of Input operation

- Create a stream object and associate it with a datasource
  - FileReader x = new FileReader(source\_file);
- Give the stream object the desired functionality through stream chaining
  - BufferedReader reader = new BufferedReader(x);
- Combine the two
  - BufferedReader reader = new BufferedReader(
    - new FileReader(source\_file));

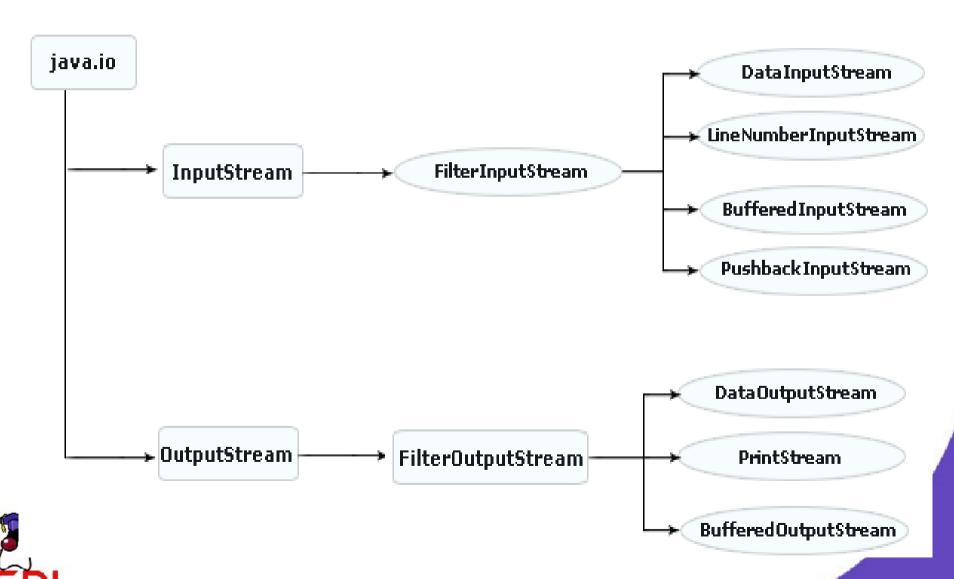


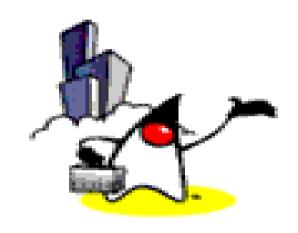
### Stream Chaining of Output operation

- Create a stream object and associate it with a data-destination
  - FileWriter x = new FileWriter(dest\_file);
- Give the stream object the desired functionality through stream chaining
  - BufferedWriter writer = new BufferedWriter(x);
- Combine the two
  - BufferedWriter writer = new BufferedWriter(
    - new FileWriter(dest\_file));



### **Filter Streams**





## **Byte 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
- Example byte stream classes
  - FileInputStream and FileOutputStream



# Example: FileInputStream & FileOutputStream

```
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);
     // More code
```

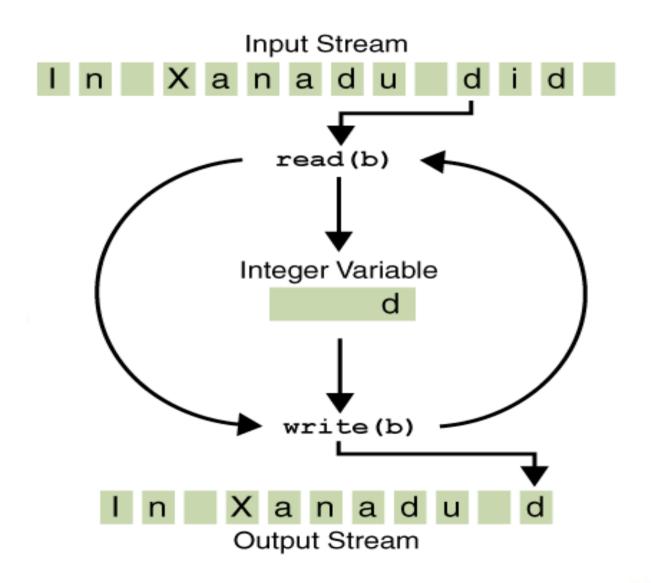


# Example: FileInputStream & FileOutputStream

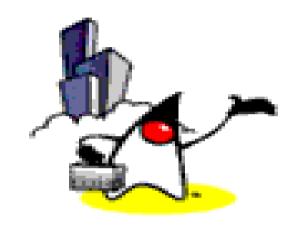
```
finally {
    if (in != null) {
        in.close();
    }
    if (out != null) {
        out.close();
     }
    }
}
```



### Simple Byte Stream input and output







### Character Stream

### **Character Stream**

- Character streams are like byte streams, but they contain 16-bit Unicode characters rather than eightbit bytes.
- They are implemented by the Reader and Writer classes and their subclasses.
- Readers and Writers support essentially the same operations as InputStreams and OutputStreams, except that where byte-stream methods operate on bytes or byte arrays, character-stream methods operate on characters, character arrays, or strings.



### **Example: FileReader & FileWriter**

```
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);
       More code
```



### **Example: FileReader & FileWriter**

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



### **Character Stream and Byte Stream**

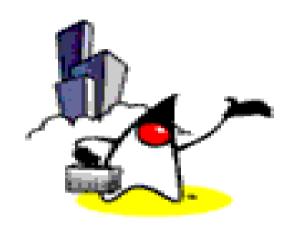
- Character streams are often "wrappers" for byte streams
- The character stream uses the byte stream to perform the physical I/O, while the character stream handles translation between characters and bytes.
  - FileReader, for example, uses FileInputStream, while FileWriter uses FileOutputStream



# Why You want to Use Character Streams over Byte Streams

- The primary advantage of character streams is that they make it easy to write programs that are not dependent upon a specific character encoding, and are therefore easy to internationalize.
- They are potentially much more efficient than byte streams.
  - The implementations of many of Java's original byte streams are oriented around byte-at-a-time read and write operations. The character-stream classes, in contrast, are oriented around bufferat-a-time read and write operations.





### **Buffered Stream**

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



### **How to create Buffered Streams?**

- A program can convert a unbuffered stream into a buffered stream using the wrapping idiom
  - A unbuffered stream object is passed to the constructor for a buffered stream class
- Example

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



### **Buffered Stream Classes**

- BufferedInputStream and BufferedOutputStream create buffered byte streams
- BufferedReader and BufferedWriter create buffered character streams



### Flushing Buffered Streams

- It often makes sense to write out a buffer at critical points, without waiting for it to fill. This is known as flushing the buffer.
- Some buffered output classes support autoflush, specified by an optional constructor argument.
  - When autoflush is enabled, certain key events cause the buffer to be flushed
  - For example, an autoflush PrintWriter object flushes the buffer on every invocation of println or format.
- To flush a stream manually, invoke its flush method
  - The flush method is valid on any output stream, but has no effect unless the stream is buffered.



#### **Line-Oriented I/O**

- Character I/O usually occurs in bigger units than single characters
  - One common unit is the line: a string of characters with a line terminator at the end
  - A line terminator can be a carriage-return/line-feed sequence ("\r\n"), a single carriage-return ("\r"), or a single line-feed ("\n").



### **Example: Line-oriented I/O**

```
File inputFile = new File("farrago.txt");
File outputFile = new File("outagain.txt");
FileReader in = new FileReader(inputFile);
FileWriter out = new FileWriter(outputFile);
BufferedReader inputStream = new BufferedReader(in);
PrintWriter outputStream = new PrintWriter(out);
String I;
while ((I = inputStream.readLine()) != null) {
  System.out.println(I);
  outputStream.println(I);
```





## Data 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



#### **DataOutputStream**

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

```
out = new DataOutputStream(
    new BufferedOutputStream(
    new FileOutputStream(dataFile)));
for (int i = 0; i < prices.length; i ++) {
    out.writeDouble(prices[i]);
    out.writeInt(units[i]);
    out.writeUTF(descs[i]);
}</pre>
```



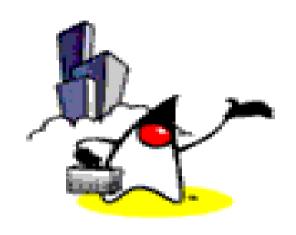
#### **DataInputStream**

- 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**

- Object streams support I/O of objects
  - 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



## 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.



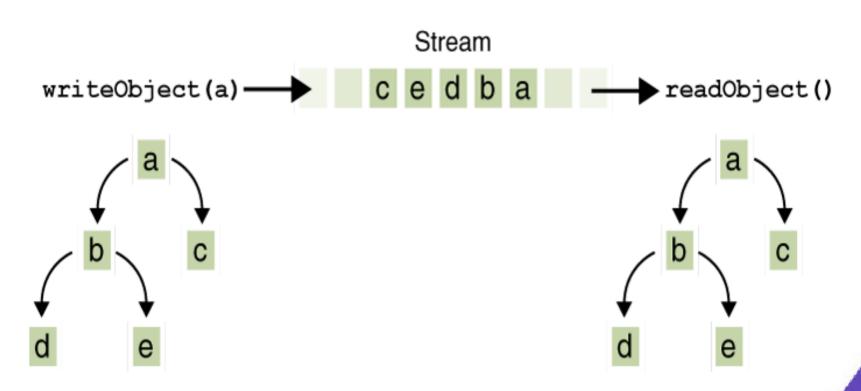
### **WriteObject**

- 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.



### I/O of multiple referred-to objects

 Object a contains references to objects b and c, while b contains references to d and e





### I/O of multiple referred-to objects

- Invoking writeObject(a) writes not just a, but all the objects necessary to reconstitute a, so the other four objects in this web are written also
- When a is read back by readObject, the other four objects are read back as well, and all the original object references are preserved.





## Standard Streams

#### Standard Streams on Java Platform

- Three standard streams
  - Standard Input, accessed through System.in
  - Standard Output, accessed through System.out
  - Standard Error, accessed through System.err
- These objects are defined automatically and do not need to be opened
- System.out and System.err are defined as PrintStream objects



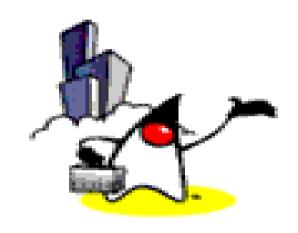


# Closing Streams

#### **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

#### The File Class

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



#### The File Class: Constructors

Has four constructors

#### A File Constructor

File (String pathname)

Instantiates a *File* object with the specified *pathname* as its filename. The filename may either be absolute (i.e., containes the complete path) or may consists of the filename itself and is assumed to be contained in the current directory.



```
import java.io.*;
2
  public class FileInfoClass {
     public static void main(String args[]) {
4
        String fileName = args[0];
5
        File fn = new File(fileName);
6
        System.out.println("Name: " + fn.getName());
7
        if (!fn.exists()) {
8
           System.out.println(fileName
9
                                + " does not exists.");
10
11 //continued...
```

```
/* Create a temporary directory instead. */
12
           System.out.println("Creating temp directory...");
13
           fileName = "temp";
14
           fn = new File(fileName);
15
           fn.mkdir();
16
           System.out.println(fileName +
17
                (fn.exists()? "exists": "does not exist"));
18
           System.out.println("Deleting temp directory...")
19
           fn.delete();
20
           //continued...
21
```



```
24
        System.out.println(fileName + " is a " +
25
                     (fn.isFile()? "file." :"directory."));
26
27
        if (fn.isDirectory()) {
28
            String content[] = fn.list();
29
            System.out.println("The content of this directory
30
            for (int i = 0; i < content.length; i++) {</pre>
36
               System.out.println(content[i]);
37
38
39
```

25 //continued...



```
System.out.println(fileName + " is " + fn.length()
47
                             + " bytes long.");
48
        System.out.println(fileName + " is " +
49
                        fn.lastModified());
50
51
        if (!fn.canWrite()) {
52
            System.out.println(fileName
53
                                 + " is not writable.");
54
55
56
57 }
```

## Thank you!

Check JavaPassion.com Codecamps!
http://www.javapassion.com/codecamps
"Learn with Passion!"

