Java I/O Stream

"Code 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?

What is an I/O Stream (or a Stream)?

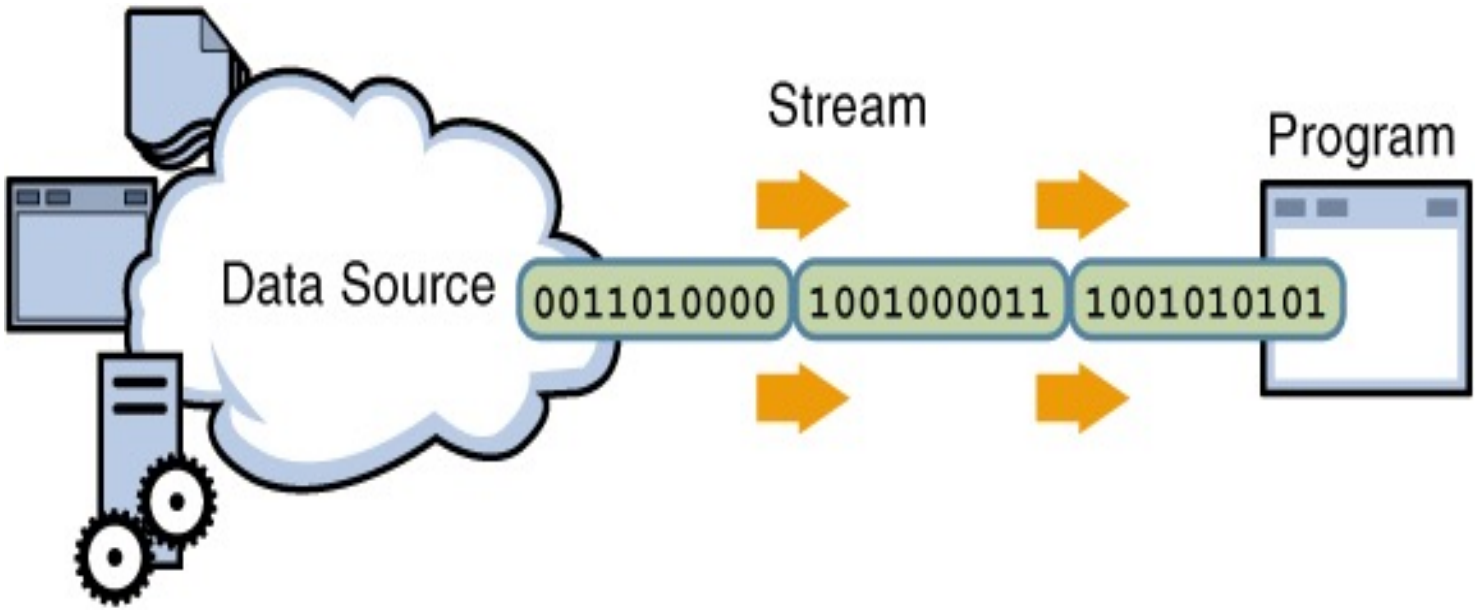
- A stream represents a sequence of data, input or output
- A stream is associated with different kinds of sources and destinations
  - Examples of sources and destinations include disk files, devices, other programs, a network socket, and memory arrays
- Streams do not store data
  - They are just programmatic wrappers over existing data sources and destinations
- Streams support different kinds of data
  - The kinds of data could be 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 of Streams

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

Input Stream from a Source

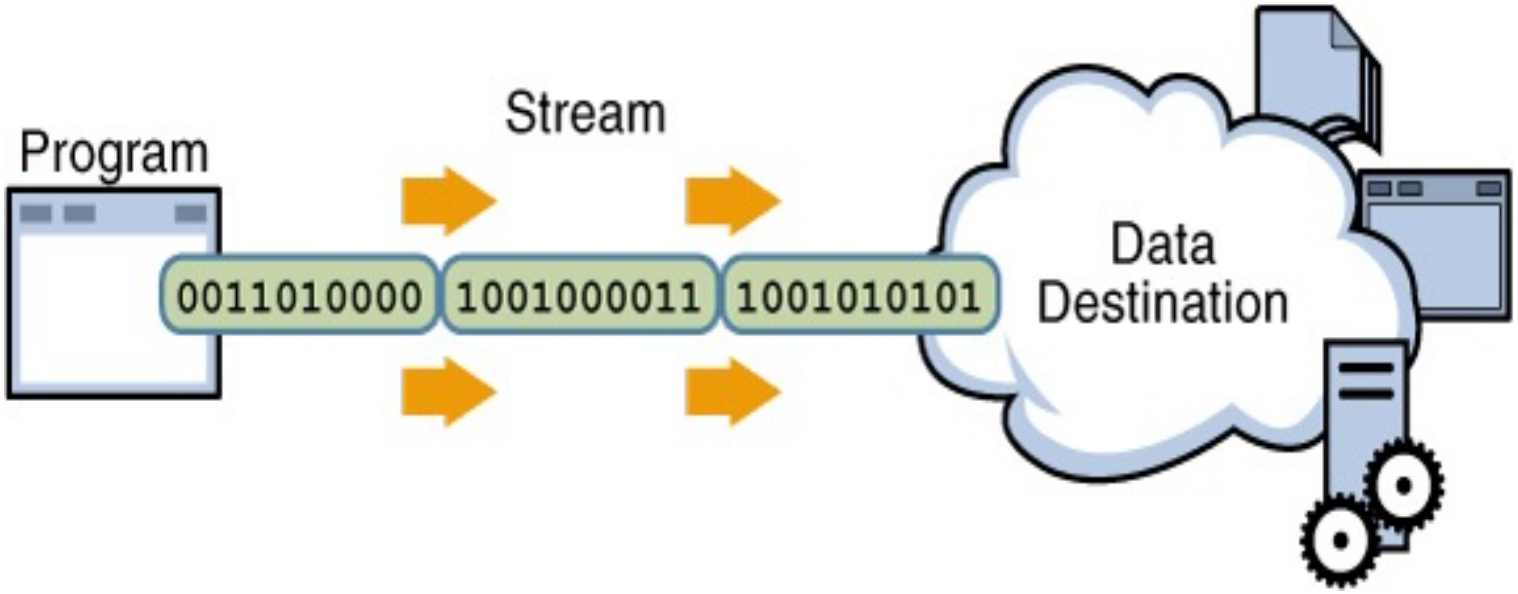
- A program uses an input stream to read data from a source, one item at a time



source:java.sun.com

Output Stream to a destination

- 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 direction
- Filter and Node Streams
  - Whether the data on a stream is manipulated/transformed (Filter Stream) or not (Node Stream)

Character and Byte Streams

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

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

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

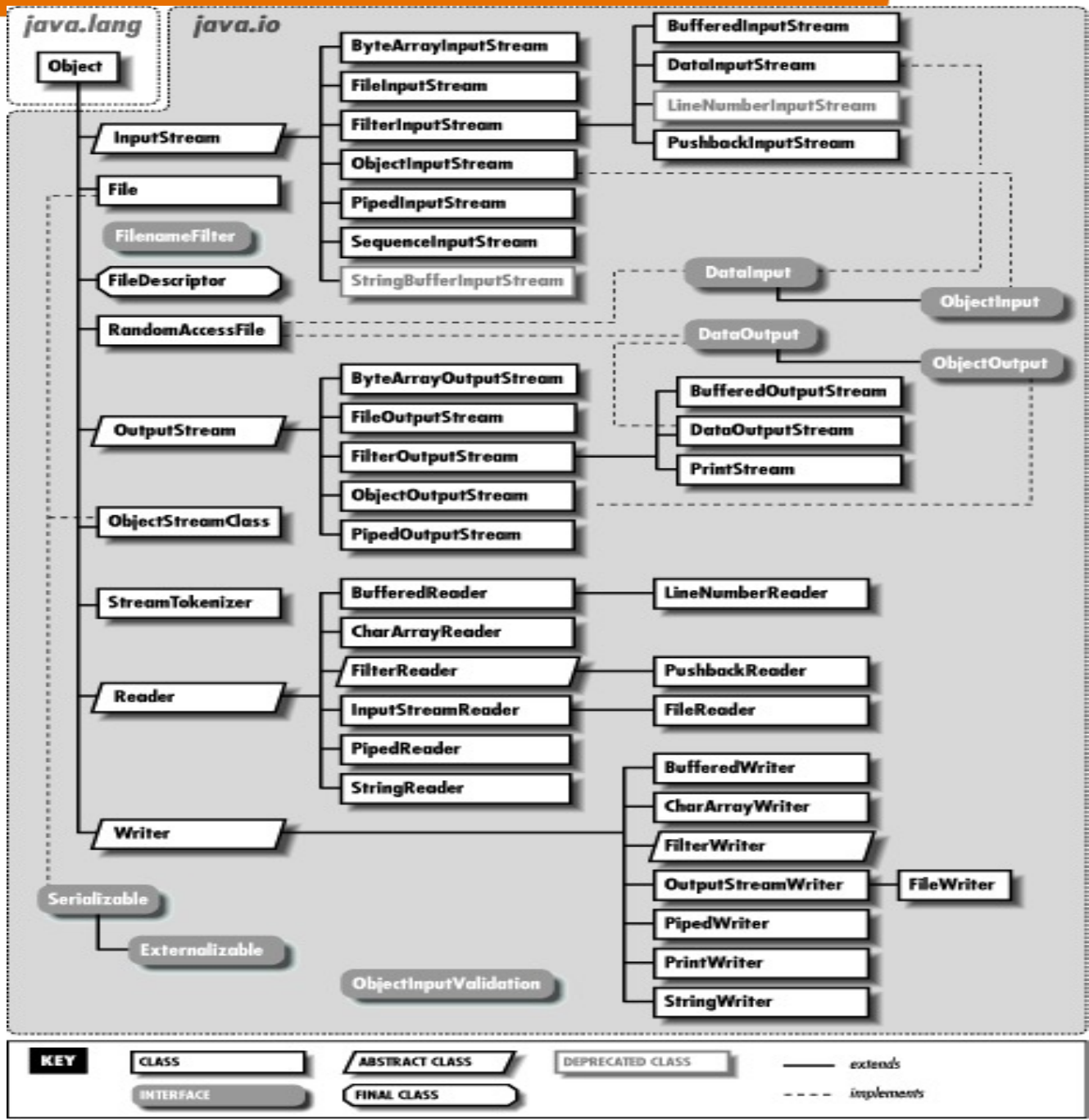
- Output or sink (destination) streams
  - Can write to these streams
  - Root classes of all output streams:
    - The *OutputStream* Class
    - The *Writer* Class

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

Streams

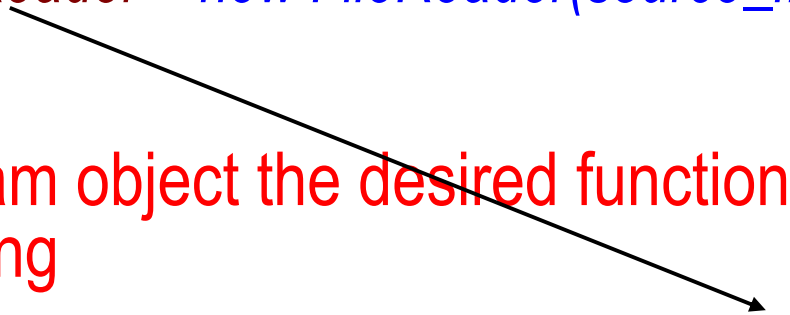

java.lang
java.io
Object
InputStream
File
FilenameFilter
FileDescriptor
RandomAccessFile
OutputStream
ObjectStreamClass
StreamTokenizer
Reader
Writer
Serializable
Externalizable
ObjectInputValidation
ByteArrayInputStream
FileInputStream
FilterInputStream
ObjectInputStream
PipedInputStream
SequenceInputStream
StringBufferInputStream
BufferedInputStream
DataInputStream
LineNumberInputStream
PushbackInputStream
DataInput
ObjectInput
DataOutput
ObjectOutput
ByteArrayOutputStream
FileOutputStream
FilterOutputStream
ObjectOutputStream
PipedOutputStream
BufferedOutputStream
DataOutputStream
PrintStream
BufferedReader
CharArrayReader
FilterReader
InputStreamReader
PipedReader
StringReader
LineNumberReader
PushbackReader
FileReader
BufferedWriter
CharArrayWriter
FilterWriter
OutputStreamWriter
PipedWriter
PrintWriter
StringWriter
FileWriter
KEY
CLASS
INTERFACE
ABSTRACT CLASS
FINAL CLASS
DEPRECATED CLASS
extends
implements

Stream Chaining

Stream Chaining of Input operation

- Create a stream object and associate it with a data-source

  FileReader fileReader = new FileReader(source_file);



- Give the stream object the desired functionality through stream chaining

  BufferedReader reader = new BufferedReader(fileReader);

- You can combine the above two

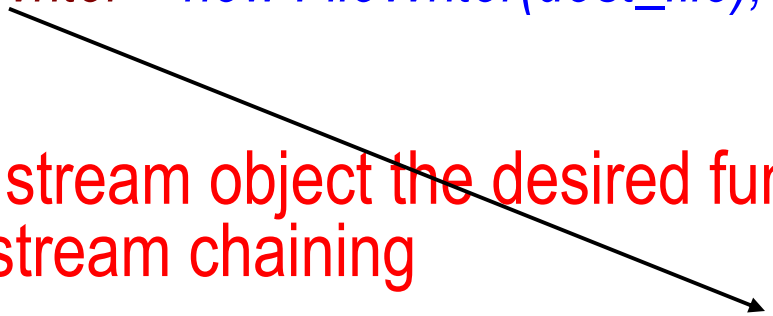
  BufferedReader reader

  = new BufferedReader(new FileReader(source_file));

Stream Chaining of Output operation

- Create a stream object and associate it with a data-destination

FileWriter fileWriter = new FileWriter(dest_file);



- Give the stream object the desired functionality through stream chaining

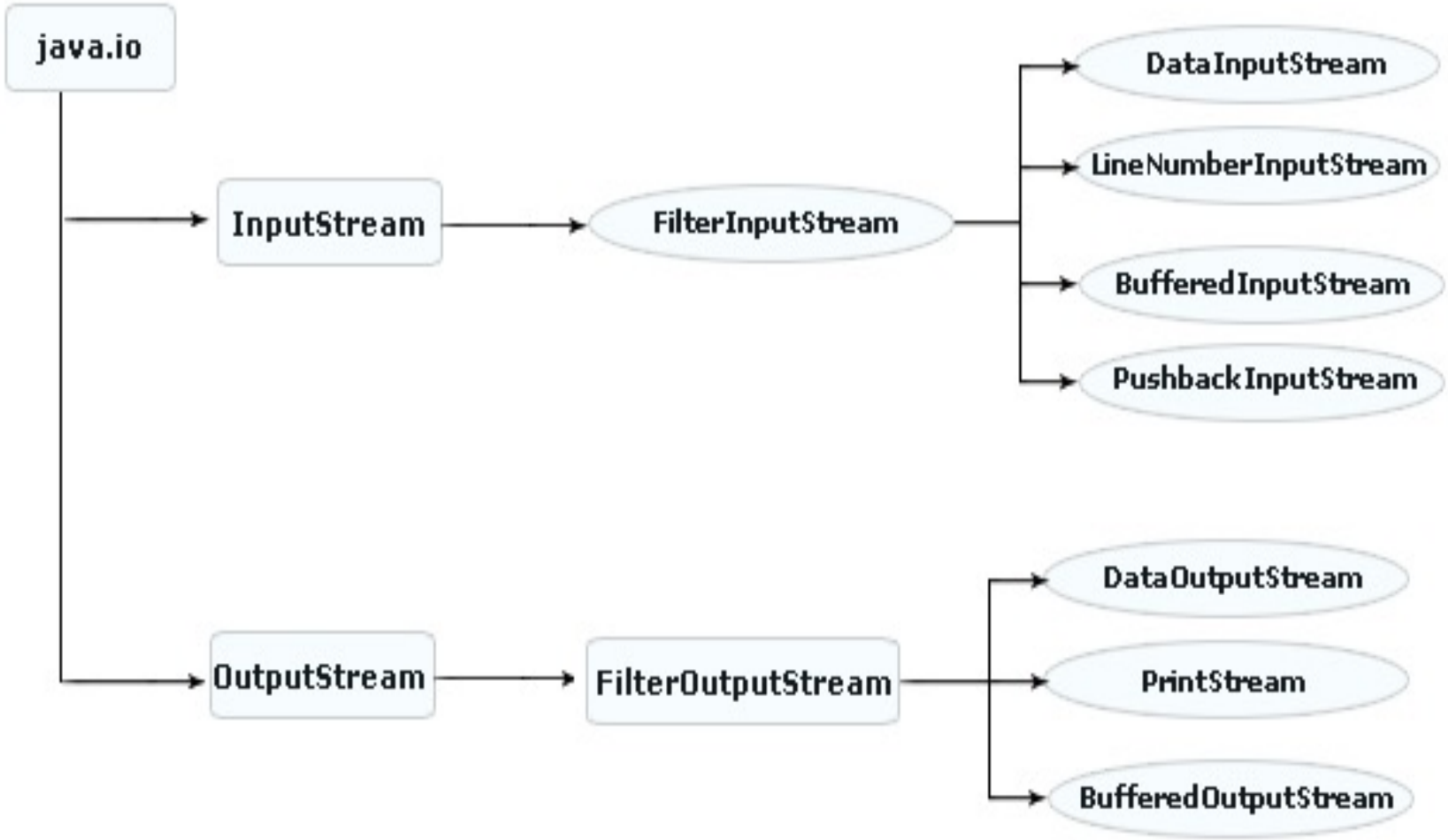
BufferedWriter writer = new BufferedWriter(fileWriter);

- Combine the two

BufferedWriter writer =

new BufferedWriter(new FileWriter(dest_file));

Filter Streams



Byte Stream

Byte Stream

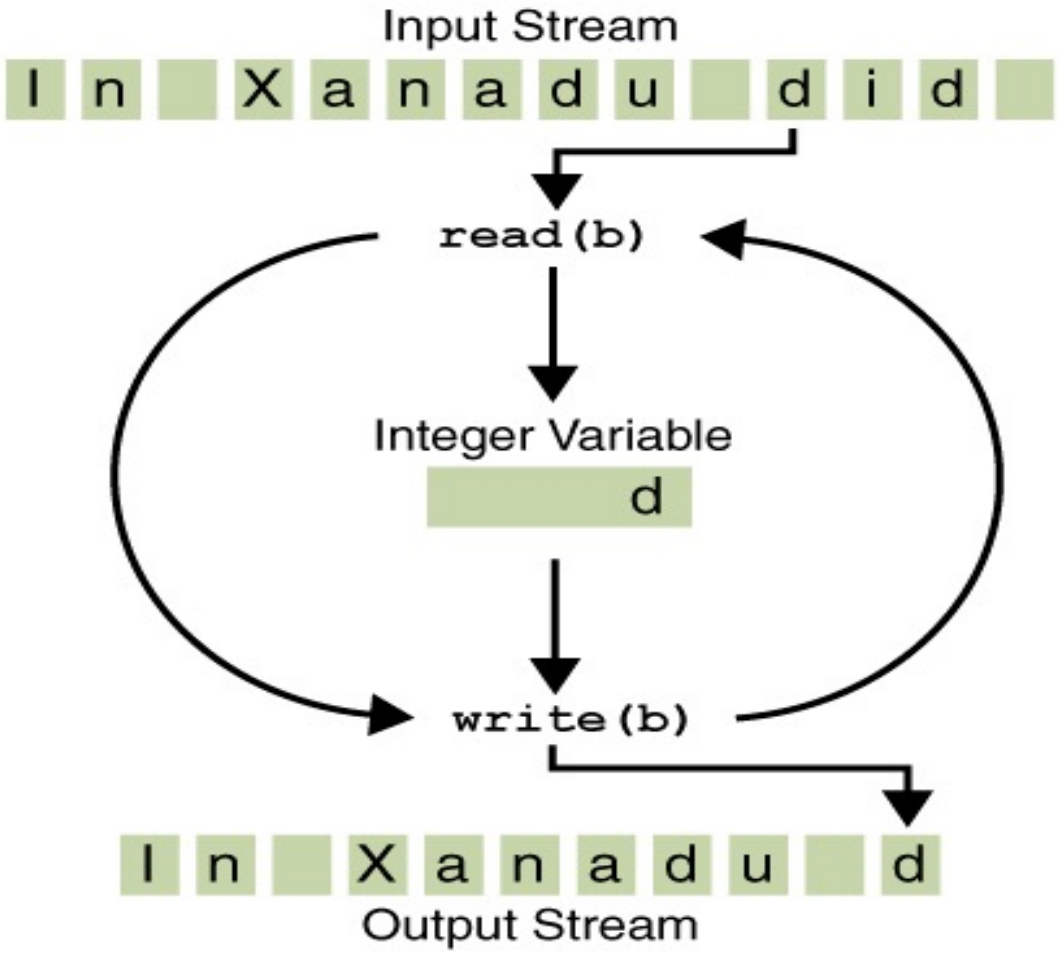
- Programs use byte streams to perform input and output of 8-bit bytes
- All byte stream classes are descended from *InputStream* and *OutputStream*
- Example byte stream classes
  - *FileInputStream* and *FileOutputStream*

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

Simple Byte Stream input and output



Lab:

Exercise 1: Byte Stream 1022_javase_iostream.zip

Character Stream

Character Stream

- Character streams are like byte streams, but they contain 16-bit Unicode characters rather than 8-bit 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);
            }
        }
        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* uses *FileInputStream*
  - *FileWriter* uses *FileOutputStream*

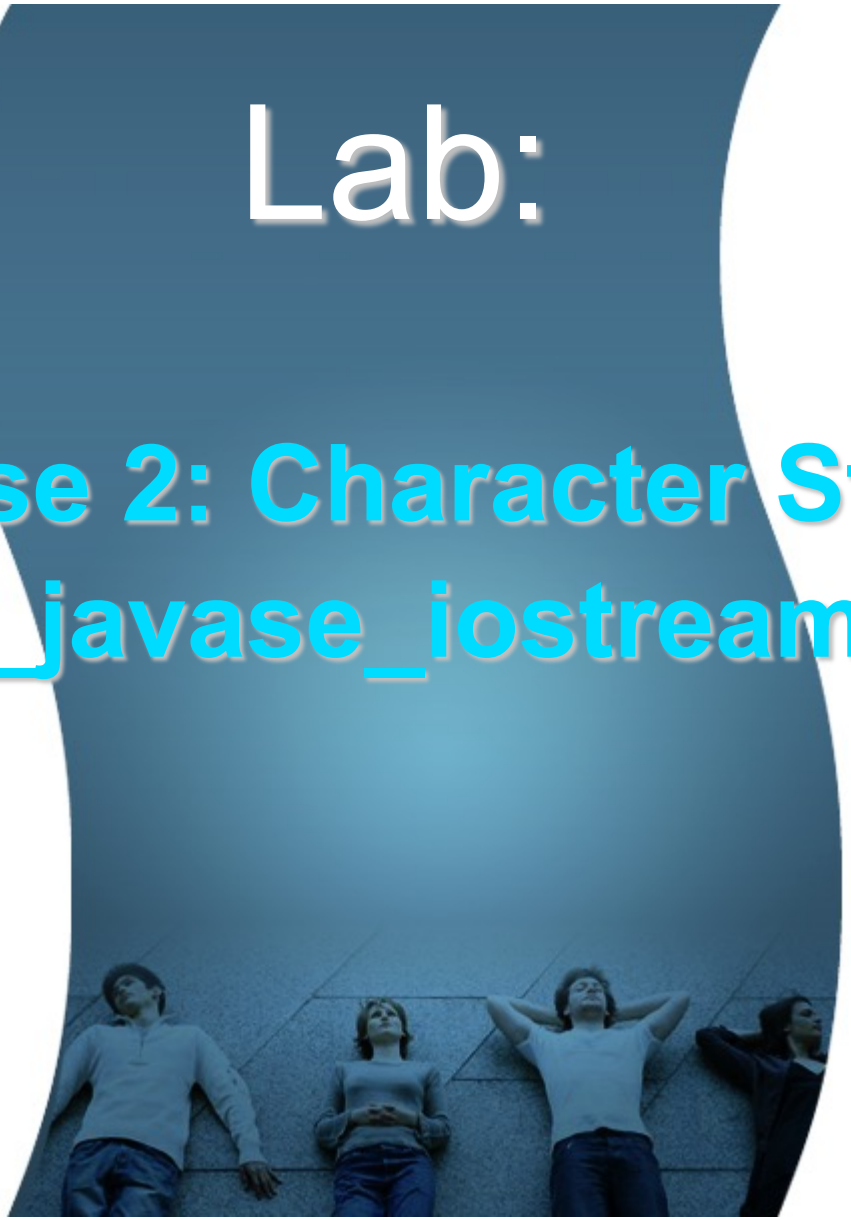
Why You want to Use Character Streams over Byte Streams

- 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 buffer-at-a-time read and write operations.

Lab:

Exercise 2: Character Stream 1022_javase_iostream.zip



Buffered Stream

Why Use 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 an unbuffered stream into a buffered stream through stream chaining
  - An unbuffered stream object is passed to the constructor of a buffered stream class

```
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 l;
while ((l = inputStream.readLine()) != null) {
    System.out.println(l);
    outputStream.println(l);
 }

in.close();
out.close();
```

Lab:

Exercise 3: BufferedReader & BufferedWriter

1022_javase_iostream.zip

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]);
}
```

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){
}
```

Lab:

Exercise 4: Data Stream
1022_javase_iostream.zip

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

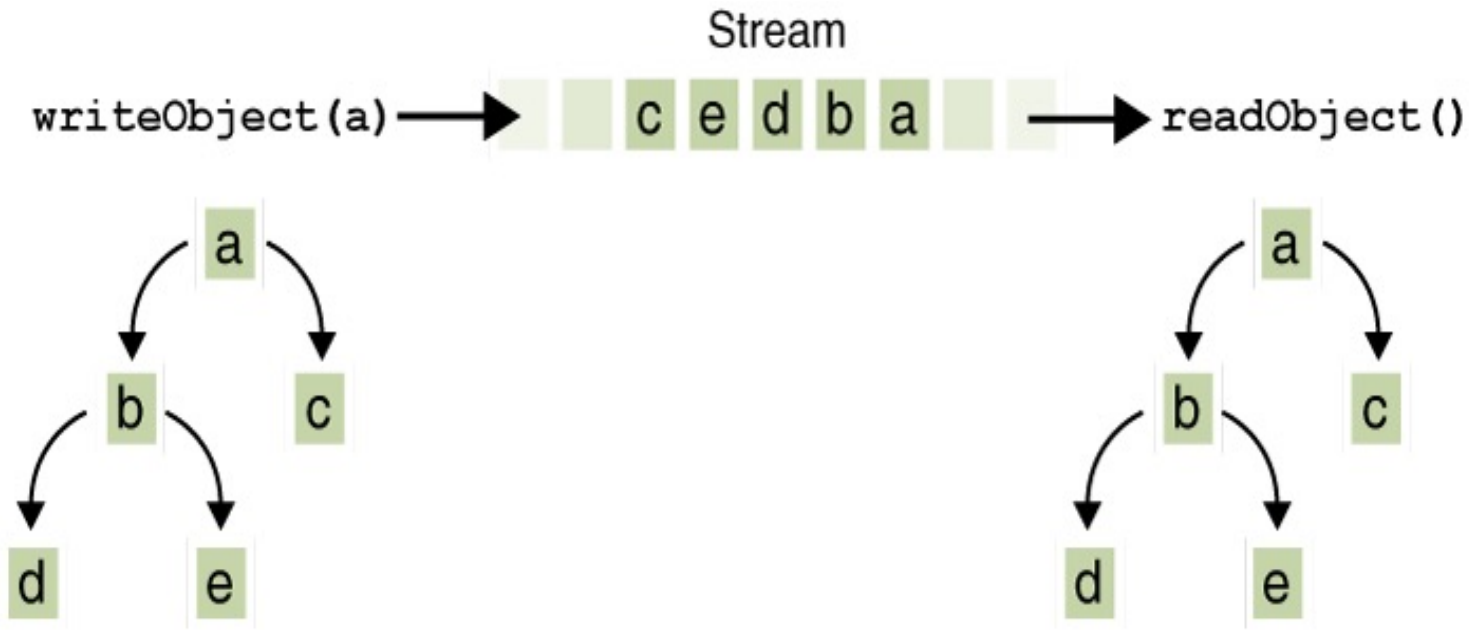
- The writeObject and readObject methods are simple to use, but they contain some very sophisticated object management logic
  - This isn't important for a class like Calendar, which just encapsulates primitive values. But many objects contain references to other objects
- If readObject is to reconstitute an object from a stream, it has to be able to reconstitute all 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.

Lab:

Exercise 5: Object Stream
1022_javase_iostream.zip

Piped Streams

Piped Streams

- A piped input stream should be connected to a piped output stream; the piped input stream then provides whatever data bytes are written to the piped output stream

- Typically, data is read from a *PipedInputStream* object by one thread and data is written to the corresponding *PipedOutputStream* by some other thread.

Lab:

Exercise 6: Piped Stream
1022_javase_iostream.zip

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.

Lab:

Exercise 7: File Handling
1022_javase_iostream.zip

Code with Passion!