CENG 1004
Introduction to Object Oriented Programming

Spring 2016

WEEK 11

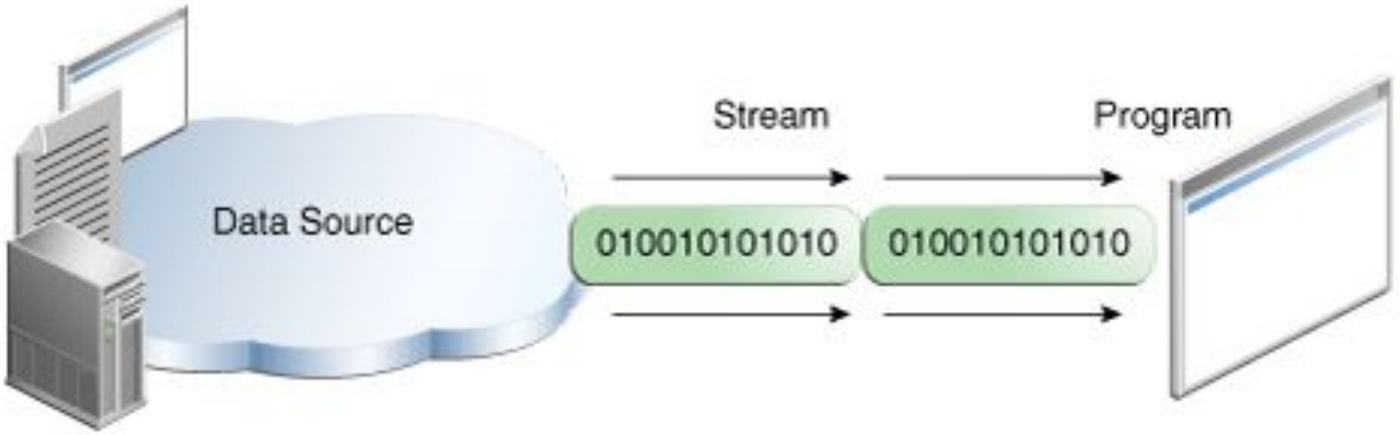
I/O Streams

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, including
  - disk files,
  - devices,
  - other programs,
  - and memory arrays.
- A stream is a sequence of data.

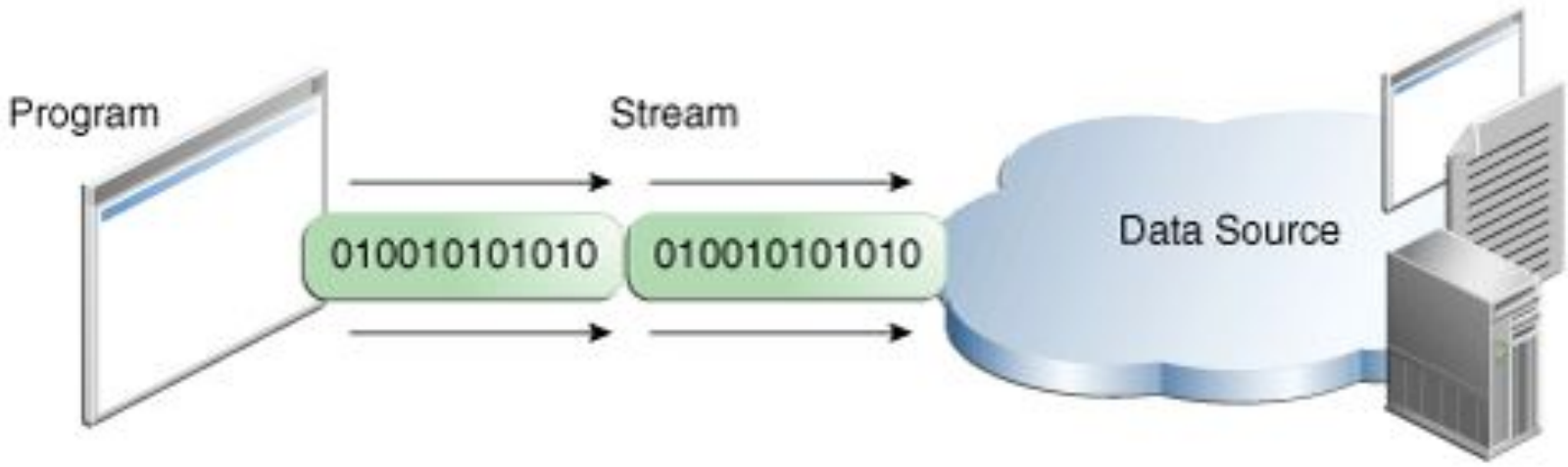
Input Stream

- 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:



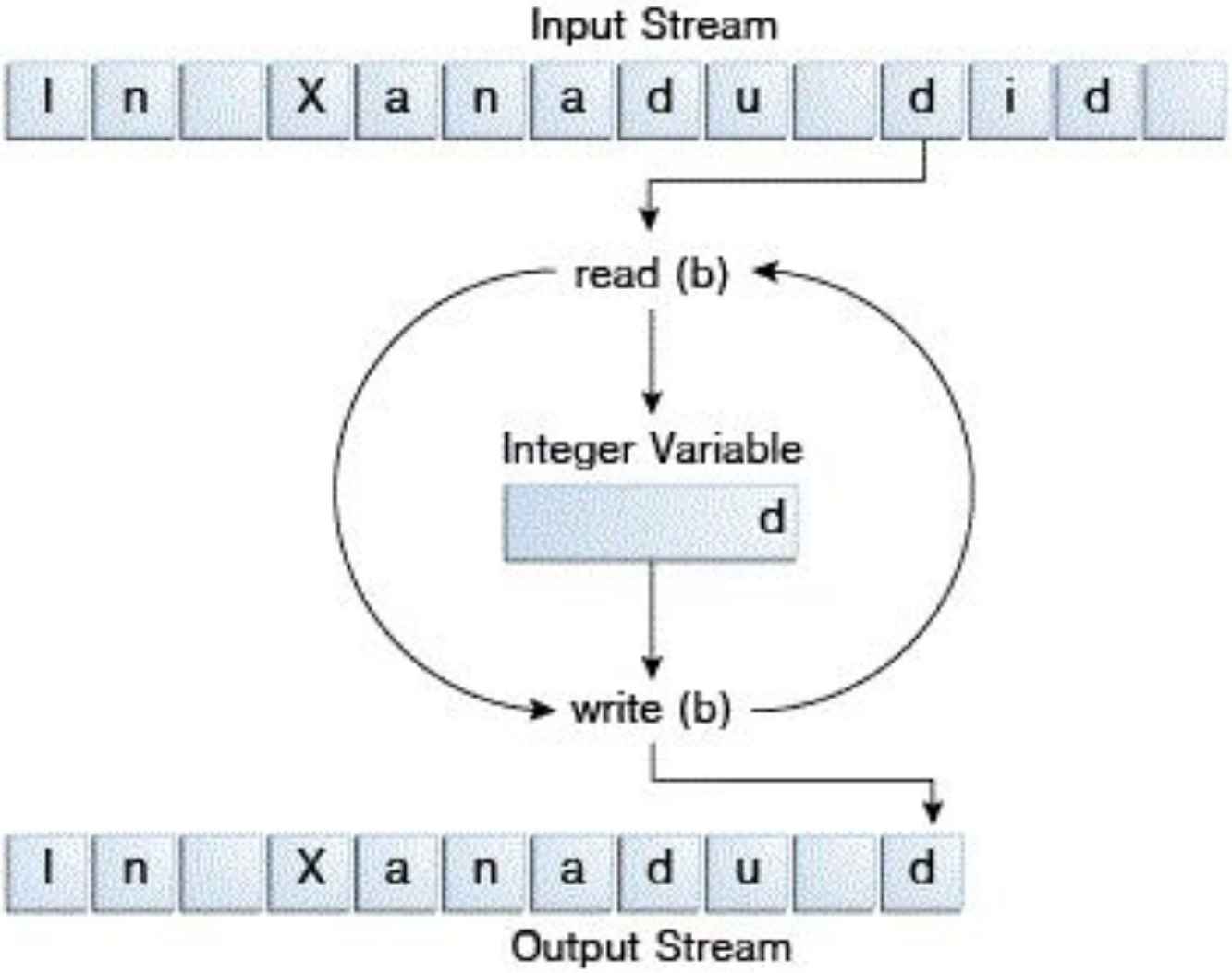
Byte Streams

- 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.
  - we'll focus on the file I/O byte streams,
    - FileInputStream and
    - FileOutputStream

CopyBytes

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

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();
            }
        }
    }
}
```

CopyBytes


Input Stream
I n X a n a d u d i d
read (b)
Integer Variable
d
write (b)
I n X a n a d u d
Output Stream

Always Close Streams

- Closing a stream when it's no longer needed is very important

- CopyBytes uses a finally block to guarantee that both streams will be closed even if an error occurs.

When Not to Use Byte Streams

- CopyBytes seems like a normal program, but it actually represents a kind of low-level I/O that you should avoid.
- Since xanadu.txt contains character data, the best approach is to use character streams
- There are also streams for more complicated data types.
- Byte streams should only be used for the most primitive I/O.
- So why talk about byte streams? Because all other stream types are built on byte streams.

Character Streams

- 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.
- A program that uses character streams in place of byte streams automatically adapts to the local character set and is ready for internationalization — all without extra effort by the programmer.
- All character stream classes are descended from Reader and Writer.

CopyCharacters

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

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 Streams that Use Byte Streams

- The most important difference is that CopyCharacters uses FileReader and FileWriter for input and output in place of FileInputStream and FileOutputStream.

- The character stream uses the byte stream to perform the physical I/O, while the character stream handles translation between characters and bytes.

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").

Line-Oriented I/O

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;

public class CopyLines {
    public static void main(String[] args) throws IOException {

        BufferedReader inputStream = null;
        PrintWriter outputStream = null;

        try {
            inputStream = new BufferedReader(new FileReader("xanadu.txt"));
            outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));

            String l;
            while ((l = inputStream.readLine()) != null) {
                outputStream.println(l);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

Buffered 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.

- 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.

Scanning

- Objects of type Scanner are useful for breaking down formatted input into tokens and translating individual tokens according to their data type.

- By default, a scanner uses white space to separate tokens.
  - White space characters include blanks, tabs, and line terminators

ScanFile

```
import java.io.*;
import java.util.Scanner;

public class ScanFile {
    public static void main(String[] args) throws IOException {

        Scanner s = null;

        try {
            s = new Scanner(new BufferedReader(new FileReader("xanadu.txt")));

            while (s.hasNext()) {
                System.out.println(s.next());
            }
        } finally {
            if (s != null) {
                s.close();
            }
        }
    }
}
```

ScanSum

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.util.Scanner;
import java.util.Locale;

public class ScanSum {
   public static void main(String[] args) throws IOException {

      Scanner s = null;
      double sum = 0;

      try {
         s = new Scanner(new BufferedReader(new FileReader("usnumbers.txt")));
         s.useLocale(Locale.US);    //new Locale("tr", "TR");

         while (s.hasNext()) {
            if (s.hasNextDouble()) {
               sum += s.nextDouble();
            } else {
               s.next();
            }
         }
      } finally {
         s.close();
      }

      System.out.println(sum);
   }
}
```

8.5
32,767
3.14159
1,000,000.1

The output string is "1032778.74159".

Formatting

- Stream objects that implement formatting are instances of either PrintWriter, a character stream class, or PrintStream, a byte stream class.

- Two levels of formatting are provided:
  - **print and println** format individual values in a standard way.
  - **format** formats almost any number of values based on a format string, with many options for precise formatting.

The print and println Methods

```
public class Root {
    public static void main(String[] args) {
        int i = 2;
        double r = Math.sqrt(i);

        System.out.print("The square root of ");
        System.out.print(i);
        System.out.print(" is ");
        System.out.print(r);
        System.out.println(".");

        i = 5;
        r = Math.sqrt(i);
        System.out.println("The square root of " + i + " is " + r + ".");
    }
}
```

The square root of 2 is 1.4142135623730951.
The square root of 5 is 2.23606797749979.

The format Method

The format method formats multiple arguments based on a format string.

```
public class Root2 {
    public static void main(String[] args) {
        int i = 2;
        double r = Math.sqrt(i);

        System.out.format("The square root of %d is %f.%n", i, r);
    }
}
```

The square root of 2 is 1.414214.

Like the three used in this example, all format specifiers begin with a % and end with a 1- or 2-character conversion that specifies the kind of formatted output being generated. The three conversions used here are:

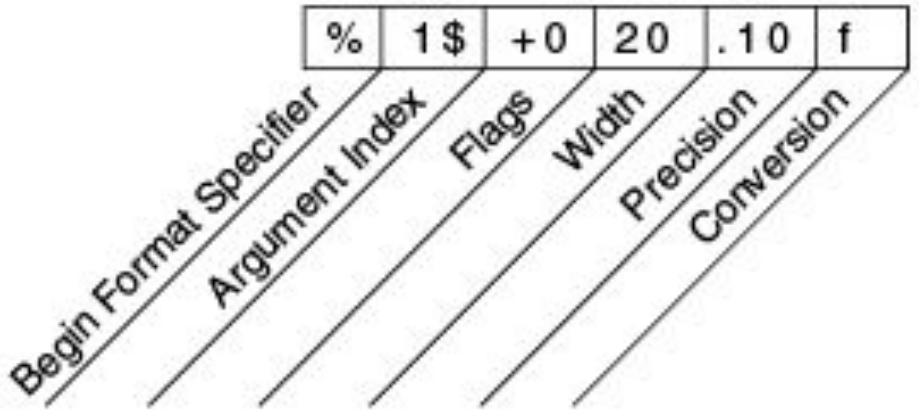
d formats an integer value as a decimal integer value.
f formats a floating point value as a decimal number.
n outputs a platform-specific line terminator.

The format Method

```
public class Format {
    public static void main(String[] args) {
        System.out.format("%f, %1$+020.10f %n", Math.PI);
    }
}
```

Here's the output:

3.141593, +00000003.1415926536



I/O from the Command Line

- A program is often run from the command line and interacts with the user in the command line environment.

- The Java platform supports this kind of interaction in two ways:
  - through the Standard Streams and
  - through the Console.

Standard Streams

- The Java platform supports 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.
- Standard Output and Standard Error are both for output;
  - having error output separately allows the user to divert regular output to a file and still be able to read error messages.

The Console

- A more advanced alternative to the Standard Streams is the Console.
- The Console is particularly useful for secure password entry.
- Before a program can use the Console,
  - it must attempt to retrieve the Console object by invoking System.console()
  - If the Console object is available, this method returns it. If System.console returns NULL

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.

- The most widely-used implementations of these interfaces, DataInputStream and DataOutputStream.

Data Streams

- Notice that DataStreams detects an end-of-file condition by catching EOFException

- Also notice that each specialized write in DataStreams is exactly matched by the corresponding specialized read.

- DataStreams uses one very bad programming technique: it uses floating point numbers to represent monetary values.
  - In general, floating point is bad for precise values. It's particularly bad for decimal fractions,
  - The correct type to use for currency values is java.math.BigDecimal.
  - Unfortunately, BigDecimal is an object type, so it won't work with data streams.

Object Streams

- Just as data streams support I/O of primitive data types, object streams support I/O of objects.

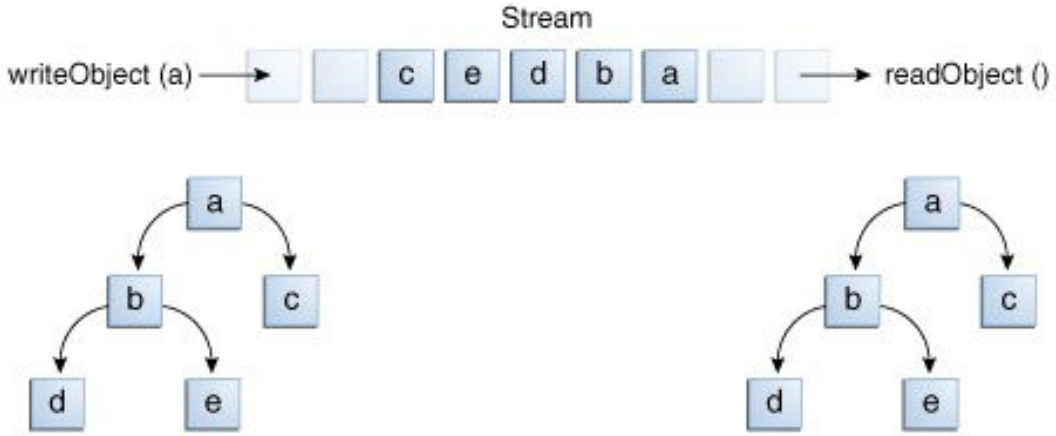
- Most, but not all, standard classes support serialization of their objects. Those that do implement the marker interface **Serializable**.

Object Streams

- The object stream classes are ObjectInputStream and ObjectOutputStream.
- These classes implement ObjectInput and ObjectOutput, which are subinterfaces of DataInput and DataOutput.
  - That means that all the primitive data I/O methods covered in Data Streams are also implemented in object streams.
- So an object stream can contain a mixture of primitive and object values.

Output and Input of Complex 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.



Multiple references

- You might wonder what happens if two objects on the same stream both contain references to a single object.

- Will they both refer to a single object when they're read back?

```
Object ob = new Object();
out.writeObject(ob);
out.writeObject(ob);
```

Multiple references

- A stream can only contain one copy of an object, though it can contain any number of references to it.

- Thus if you explicitly write an object to a stream twice, you're really writing only the reference twice.

Multiple references

- Each writeObject has to be matched by a readObject, so the code that reads the stream back will look something like this:

```
Object ob1 = in.readObject();
Object ob2 = in.readObject();
```

- This results in two variables, ob1 and ob2, that are references to a single object.

References

- http://math.hws.edu/javanotes/
- https://docs.oracle.com/javase/tutorial/essential/io/streams.html