CSSE 220 Day 28

Non-text Files, Reading and Writing Objects
Work on Spellchecker Project

CSSE 220 Day 28

- Turn in last written problems now.
- Mini-project is due at the beginning of Day 30 class (no late days)
  - Just before your presentation, we will randomly choose which of your team members will present, so everyone should be prepared to do it.
  - Commit an outline of your presentation to your team repository by beginning of class on Thursday.
  - You will use my machine for the demo (to help keep transition time down), so make sure your repository is populated by 7am on Friday
  - There will be time in class to work with your team today and tomorrow. Do not miss it!
- Questions?
- Today:
  - Random access files and serialization
  - Work on Spellchecker

CSSE 220 Day 28

- Note: If you like looking at sorting code and animations, there are yet more at:
- http://www.brian-borowski.com/Sorting/

Course Evaluations

- I will provide some class time on Thursday for filling out the evaluation forms
- I recommend that you wait until then to do them, so you'll be able to comment on the full course, including your project experience.

Java I/O (Input and Output) 1

- Back In the Day [TM]
    - I/O only involved a few possible sources/destinations
    - terminal, printer, card reader, hard disk
    - Typically there were separate sets of functions for each type of source or destination.
- Now there are many more sources/destinations
    - including network locations.
    - and we recognize that most of the I/O functions are common to all sources/destinations
- In order to make **all** I/O more flexible and adaptable in Java, **simple** I/O is more complex than in some other languages.

Java I/O (Input and Output) 2

- What is a Stream?
  - An abstract representation of information flow that is independent of the source and/or destination.
- A stream is One-Way
  - Either an Input Stream or an Output Stream
- InputStream
  - Subclasses include FileInputStream, ObjectInputStream, AudioInputStream.
  - A socket has a **getInputStream** method that lets us get info from a network connection.
  - **System.in** is an InputStream
- OutputStream
  - Subclasses include FileOutputStream, ObjectOutputStream.
  - A **PrintStream** is a specialized OutputStream with characteristics suitable for standard output.
  - **System.out** is a PrintStream.

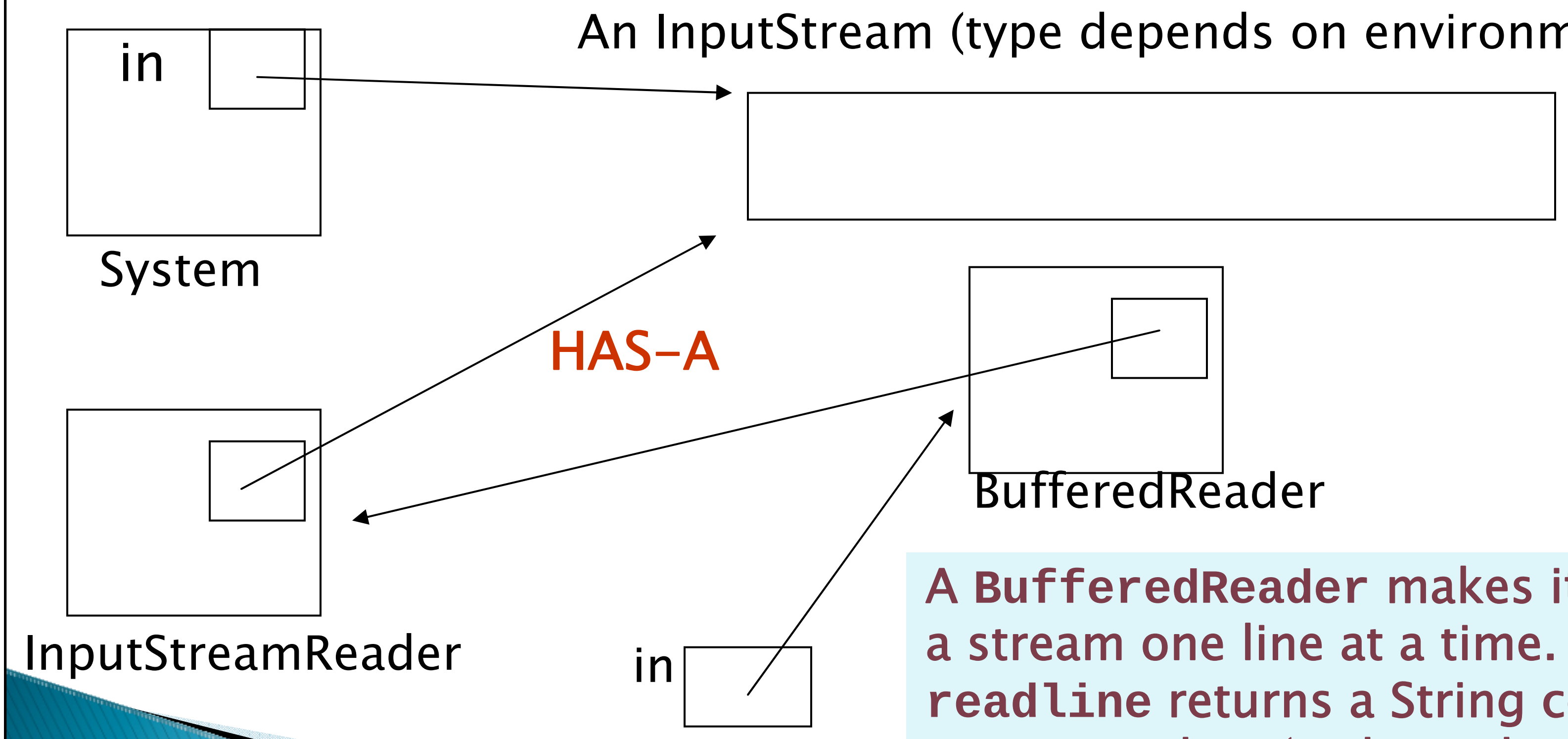
Java I/O (Input and Output) 3

- Three pre-defined streams
  - System.in (an InputStream)
  - System.out (a PrintStream)
  - System.err (a PrintStream)
- Streams are byte-oriented.
  They read or write bytes or arrays of bytes.
- Readers and Writers are character-oriented, they read or write characters or arrays of characters.
- Examples of Reader classes:
  - InputStreamReader, BufferedReader, FileReader, PushBackReader, StringReader.
- Examples of Writer classes:
  - OutputStreamWriter, PrintWriter, BufferedWriter, StringWriter

Reader Construction – From `System.in`

Line-at-a-time input from the standard input stream **`System.in`**

```
BufferedReader in = new BufferedReader(
                         new InputStreamReader( System.in ) );
```



An InputStream (type depends on environment)

in

System

HAS-A

BufferedReader

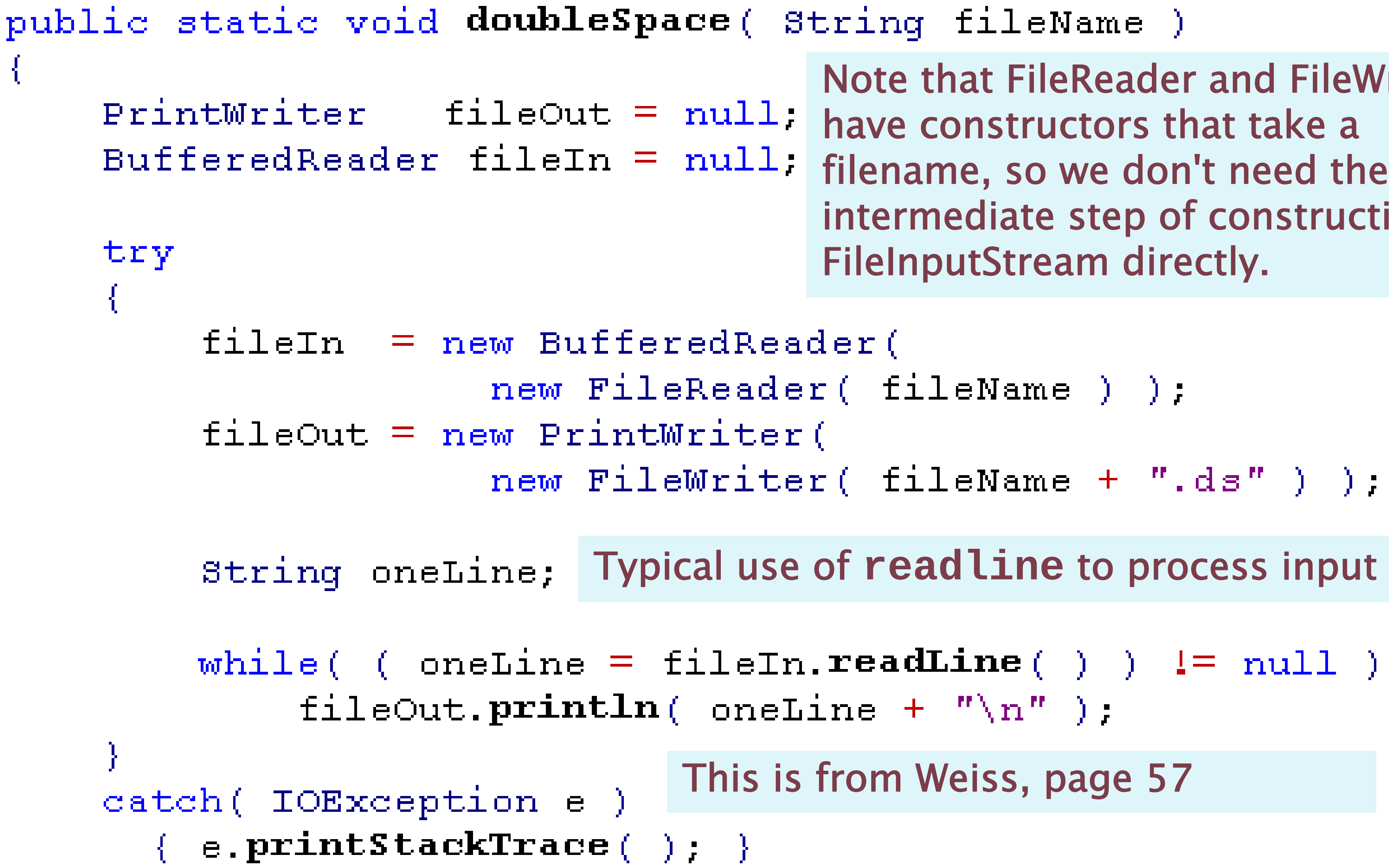
InputStreamReader

in

A **`BufferedReader`** makes it easy to read a stream one line at a time. Each call to **`readline`** returns a String containing the next input line (without the end-of-line character).

Reader/Writer Construction – From files

I/O to/from files using a BufferedReader and a PrintWriter.



```
public static void doubleSpace( String fileName )
{
    PrintWriter    fileOut = null;
    BufferedReader fileIn = null;

    try
    {
        fileIn  = new BufferedReader(
                      new FileReader( fileName ) );
        fileOut = new PrintWriter(
                      new FileWriter( fileName + ".ds" ) );

        String oneLine;

        while( ( oneLine = fileIn.readLine( ) ) != null )
            fileOut.println( oneLine + "\n" );
    }
    catch( IOException e )
      { e.printStackTrace( ); }
```

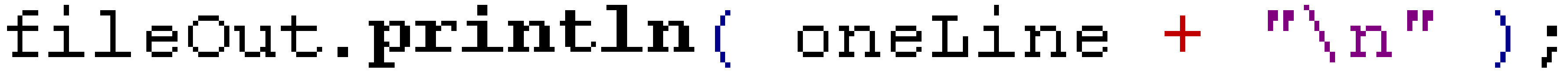
Note that FileReader and FileWriter have constructors that take a filename, so we don't need the intermediate step of constructing an FileInputStream directly.

Typical use of **readline** to process input

This is from Weiss, page 57

Weiss's one bad idea in that example

Can you see what is not so good about the code on the previous slide?



```
fileOut.println( oneLine + "\n" );
```

What should we do instead?

```
System.getProperty("line.separator");
```

Reading and Writing Objects

- We'd like to be able to write objects to a file, then read them back in later.
- Java (transparently to the user) writes type information along with the data.
- Reading the object in will recover its type information.

Issues with reading/writing Objects

- Objects can contain references to other objects.
  - Writing out the actual reference (a memory address) would be meaningless when we try to read it back in.
- Several objects might have references to the same object.
  - We do not want to write out several copies of that object to the file.
  - If we did, we might read them back in as if they were different objects.

Solution: Object Serialization

- The objects that we write/read must implement the **Serializable** interface (which has no methods).
- Objects are written to an ObjectOutputStream.
- An example should help you see how it works.

Demo

1. Paint, with drawings you can save, then clear, then load, and **undo.**

   Clearly not using images.

2. A savings account example

3. Why the Paint demo works

Example: 1. Serializable classes

```
class Person implements Serializable{
   private String name;
   public Person (String name) {
         this.name=name; }
   }

class Account implements Serializable {
   private Person holder;
   private double balance;
   public Account(Person p, double amount) {
     holder=p;
     balance=amount;
   }
}

class SavingsAccount extends Account implements Serializable {
   private double rate;
   public SavingsAccount(Person p, double amount, double r) {
     super(p,amount);
     rate=r;
}
```

Note that an **Account** HAS-A **Person**

Example: 2. Definitions and Output

```
public static void main(String [] args) {
  try {
    Person fred = new Person("Fred");
    Account general = new Account(fred, 110.0);
    Account savings = new SavingsAccount(fred, 500.0, 6.0);

    ObjectOutputStream oos = new ObjectOutputStream(
                 new FileOutputStream("Objects.dat"));
    oos.writeObject(general);
    oos.writeObject(savings);
    oos.close();
```

- In addition to **writeObject( )**, the ObjectOutputStream class provides methods for writing primitives, such as **writeDouble( )** and **writeInt( )**. writeObject( ) calls these when needed.

Example: 3. Input Serialized Objects

```
ObjectInputStream ois =
       new ObjectInputStream(
              new FileInputStream("Objects.dat"));
 Account aGeneral = (Account)ois.readObject();
 Account aSavings = (Account)ois.readObject();
```

- We must read the objects in the same order as they were written.
- Both objects that are read are assigned to variables of the type **Account**, even though one should have been written out as a **SavingsAccount.**
- We will check to make sure it was read correctly.

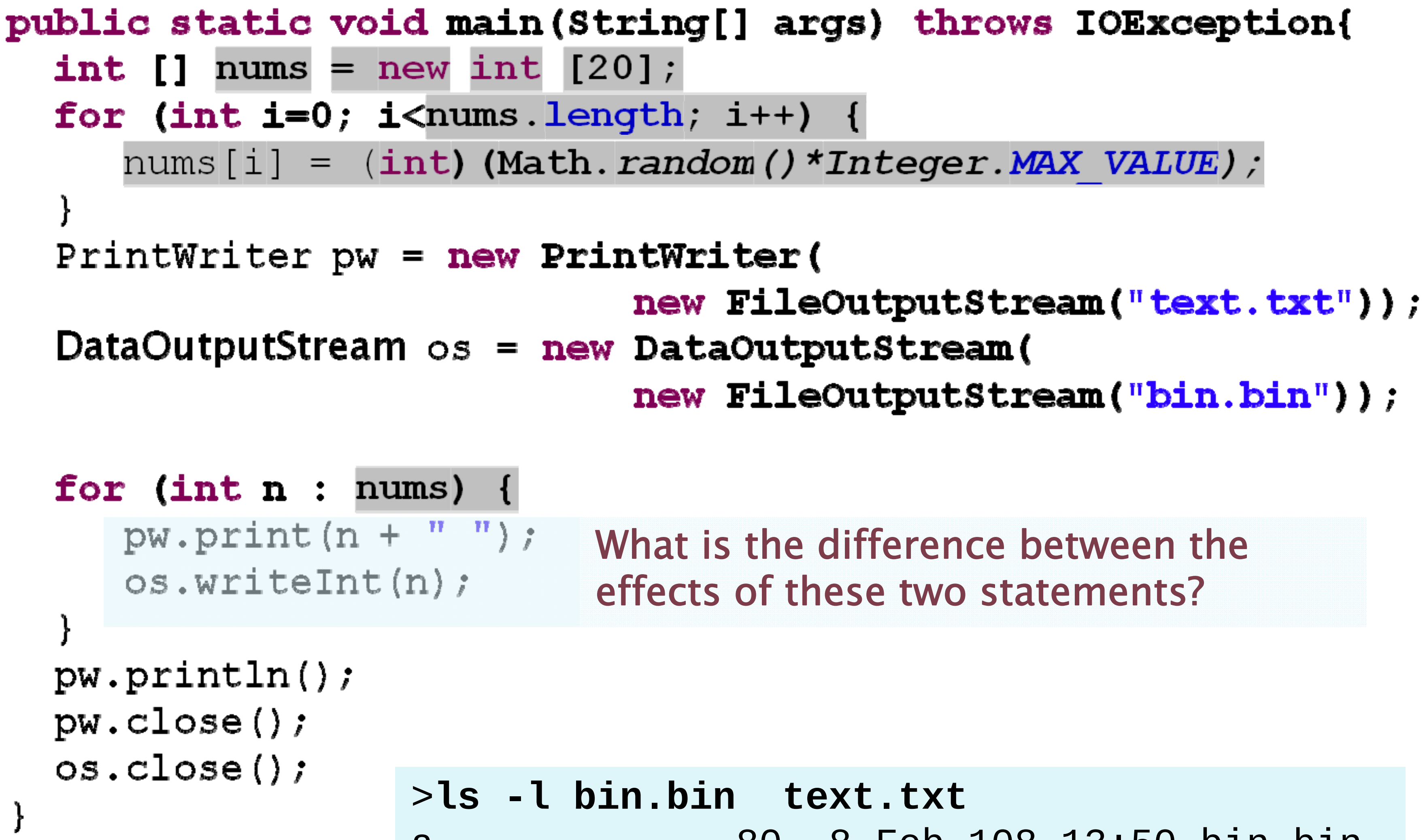
Example: 4. Check the Objects

```
  if (aGeneral instanceof SavingsAccount)
      System.out.println("aGeneral is a SavingsAccount");
  else if (aGeneral instanceof Account)
      System.out.println("aGeneral is an Account");
  if (aSavings instanceof SavingsAccount)
      System.out.println("aSavings is a SavingsAccount");
  else if (aSavings instanceof Account)
      System.out.println("aSavings is an Account");
  if (aGeneral.holder == aSavings.holder)
     System.out.println("The account holder, fred, is shared");
  else
     System.out.println("Account holder, fred, was duplicated");
  ois.close();
}catch (IOException ioe) {
     ioe.printStackTrace();
}catch (ClassNotFoundException cnfe) {
     cnfe.printStackTrace();
}
```

Output:

```
aGeneral is an Account
aSavings is a SavingsAccount
The account holder, fred, is shared
```

Text Files vs Binary files



```
public static void main(String[] args) throws IOException{
  int [] nums = new int [20];
  for (int i=0; i<nums.length; i++) {
     nums[i] = (int)(Math.random()*Integer.MAX_VALUE);
  }
  PrintWriter pw = new PrintWriter(
                           new FileOutputStream("text.txt"));
  DataOutputStream os = new DataOutputStream(
                           new FileOutputStream("bin.bin"));

  for (int n : nums) {
     pw.print(n + " ");
     os.writeInt(n);
  }
  pw.println();
  pw.close();
  os.close();
}
```

What is the difference between the effects of these two statements?

```
>ls -l bin.bin  text.txt
a-----          80  8-Feb-108 13:50 bin.bin
a-----         211  8-Feb-108 13:50 text.txt
```

UNIX output format is more compact than MSDOS.

Random Access Files

Streams provide easy sequential access to a file, but sometimes you want to have random access; for example a database program certainly needs to be able to go directly to a particular location in the file.

```
import java.io.*;
public class RandomAccess {
  public static void main(String [] args) {
   try {
    RandomAccessFile raf = new RandomAccessFile("random.dat", "rw");
    for (int i=0; i<10; i++)
      raf.writeInt(i);
    raf.seek(20);
    int number = raf.readInt();
    System.out.println("The number starting at byte 20 is " + number);
    raf.seek(4);
    number = raf.readInt();
    System.out.println("The number starting at byte 4 is "  + number);
    raf.seek(5);
    number = raf.readInt();
    System.out.println("The number starting at byte 5 is "  + number);

    raf.close();
   }catch (IOException e) {
      e.printStackTrace();
   }
  }
}
```

writeInt ?

Note that we are reading and writing numbers in their internal (binary) representation, not in their text (human-readable) representation.

This example is adapted from Art Gittleman, *Advanced Java:Internet Programming*, page 16