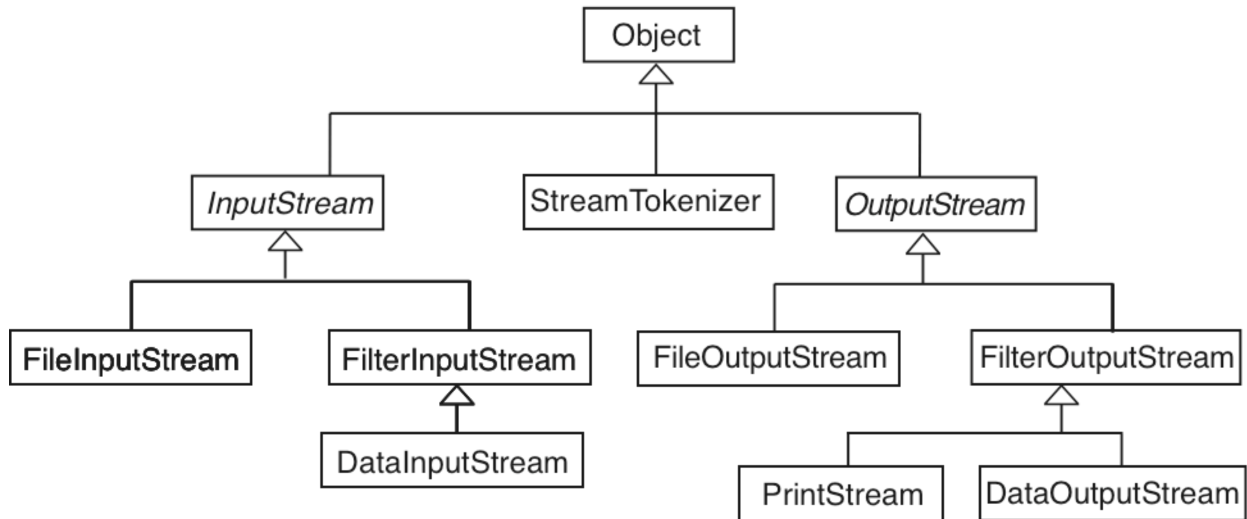
Input and Output

A stream is an ordered sequence of data in the form of bytes or characters.

Byte-Oriented Streams



InputStream

- An abstract class.
- An InputStream has a producer at one end placing bytes onto the stream that are read by the methods in the subclasses of InputStream
- System.in is an InputStream object.
- Methods

  public int read() **throws** IOException

  Reads one byte and stores it as an integer.
  End-of-File signaled by -1.
  Can be cast as character or byte.

public int read(**byte** [] ba) **throws** IOException
Fills an *existing* array with bytes read.
Returns the number of bytes read.
Returns -1 at the end of the stream.

public long skip(**long** n) **throws** IOException
Skips next *n* bytes.

public void close() **throws** IOException
Returns resources to operating system.

OutputStream

- Abstract class.
- An OutputStream has a consumer at one end removing bytes from the stream that are written by methods in subclasses of OutputStream
- System.out and System.err are objects from PrintStream, a subclass of OutputStream.
- Methods

public void write(**int** b) **throws** IOException
Writes one byte stored as an integer.

public void write(**byte** [] ba) **throws** IOException
Writes an array of bytes.

public void write(**byte** [] ba, **int** p, **int** k) **throws** IOException
Writes *k* bytes starting at position *p* in array of bytes.

public void flush() **throws** IOException
Flushes output buffer maintained by OS.

public void close() **throws** IOException
Returns resources to operating system.

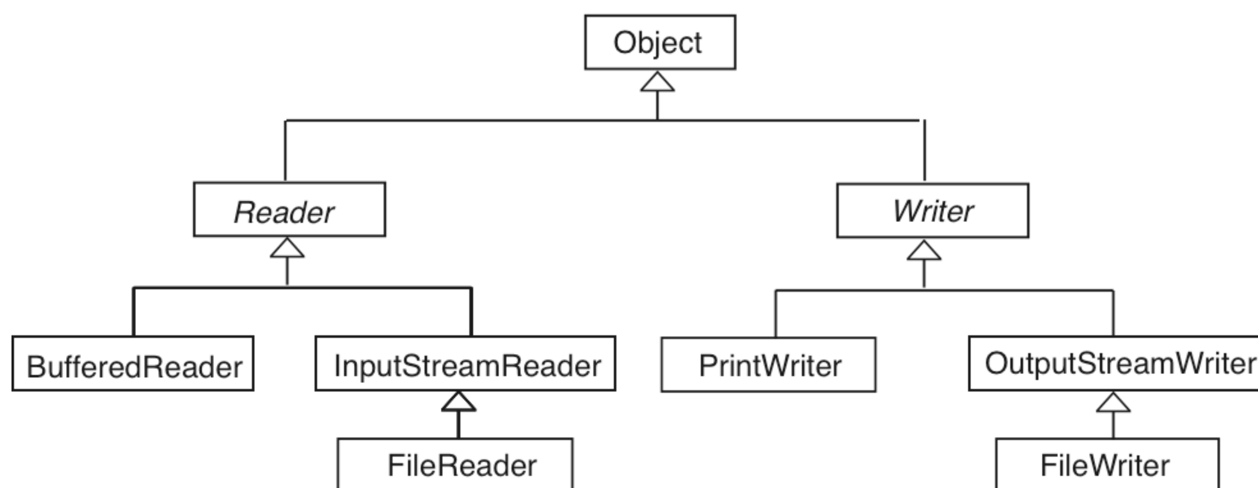
 Copyright 2005 by Ken Slonneger

Read Characters and Echo to Display

```
import java.io.*;

public class ReadChars
{
    public static void main (String [] args)
    {
        System.out.println("Enter text terminated by a # character");
        try
        {
            readChars();
            System.out.println();
        }
        catch (IOException e)
        {   System.out.println("Caught an IOException");   }
    }


    static void readChars() throws IOException
    {
        int ch = System.in.read();
        while (ch != '#')
        {
            System.out.write(ch);
            ch = System.in.read();
        }
    }
}
```

Character Streams

Classes for manipulating streams of characters (**char**).



The methods in the abstract classes Reader and Writer parallel the methods in InputStream and OutputStream, respectively.

Reader

public int read() **throws** IOException
public int read(**char** [] ca) **throws** IOException
public long skip(**long** n) **throws** IOException
public void close() **throws** IOException

Writer

public void write(**int** c) **throws** IOException
public void write(**char** [] ca) **throws** IOException
public void write(**char** [] ca, **int** p, **int** k) **throws** IOException
public void flush() **throws** IOException
public void close() **throws** IOException
public void write(**String** s) **throws** IOException

 Copyright 2005 by Ken Slonneger

Classifying Streams

Streams can be distinguished by the roles they play.

Physical Streams

- Tied to some IO device that is producing or consuming bytes or characters.

    System.in and System.out

    FileInputStream and FileOutputStream

    FileReader and FileWriter

Virtual Streams

- Add functionality to already existing streams by wrapping them in a new object.

    BufferedReader

    DataInputStream and DataOutputStream

    PrintStream and PrintWriter

    InputStreamReader and OutputStreamWriter

Converting Between Byte and Char Streams

A constructor for InputStreamReader takes an InputStream as a parameter and produces a Reader object.

Since System.in is an InputStream,

```
new InputStreamReader(System.in)
```

is a Reader (a stream of characters).

A constructor for OutputStreamWriter takes an OutputStream as a parameter and produces a Writer object.

Since System.out is an OutputStream,

```
new OutputStreamWriter(System.out)
```

is a Writer (a stream of characters).

BufferedReader

Takes a character stream (a Reader) and creates a new (filtered) stream with additional functionality:

```
public final String readLine() throws IOException
```

This method returns the value **null** when the end of the stream is reached.

Perform Input One Line at a Time

```
import java.io.*;
public class ReadLines
{
   public static void main(String [] args)
   {
      BufferedReader br = new BufferedReader(
                              new InputStreamReader(System.in));
      System.out.println("Enter text (empty line at end)");
      try
      {  String str = br.readLine();
         while (!str.equals(""))
         {  System.out.println (str);
            str = br.readLine();
         }
      }
```

 Copyright 2005 by Ken Slonneger

```
      catch (IOException e)
      {  System.out.println("Caught an IOException");  }
   }
}
```

File IO

File streams also come in two varieties,

Character streams: FileReader and FileWriter

Byte streams: FileInputStream and FileOutputStream

The behavior of these streams was created by overriding the basic methods for Readers and Writers (and InputStreams and OutputStreams):

read, write, flush, close, etc.

Each class has three constructors—we use the ones that take a string as a parameter (the filename).

Copy a Text File

Text files are best handled as character streams.

Copy a source file into a destination file by reading into an array of characters.

```
import java.io.*;

public class CopyFile
{
   public static void main(String [] args)
   {
      char [] chars = new char [128];
```

```
        BufferedReader br =
                new BufferedReader(
                        new InputStreamReader(System.in));
        try
        {
          System.out.print("Enter a source filename: ");
          String sourceName = br.readLine().trim();

          System.out.print("Enter a target filename: ");
          String targetName = br.readLine().trim();

          FileReader istream = new FileReader(sourceName);
          FileWriter ostream = new FileWriter(targetName);

          int count = istream.read(chars);
          while (count != -1)      // read returns -1 at end of stream
          {
            ostream.write(chars, 0, count);
            count = istream.read(chars);
          }
          istream.close();
          ostream.close();        // flushed the output stream
        }
        catch (IOException e)
        {  System.out.println(e)  }
        System.out.println("Done");
    }
}
```

 Copyright 2005 by Ken Slonneger

Copy a Text File by Lines

Using BufferedReader and PrintWriter allows us to program at a higher level of abstraction.

This solution uses command-line arguments to provide the names of the two files. Observe how the program reports a misuse of this syntax.

The BufferedReader method *readLine* returns the value **null** at the end of its stream.

A PrintWriter responds to the same two methods, *print* and *println*, that a PrintStream recognizes.

```
import java.io.*;

public class CopyLines
{
    public static void main(String [] args)
    {
        if (args.length != 2)
        {
            System.out.println(
                        "Usage: java CopyLines source target");
            return;
        }
        String source = args[0];
        String target = args[1];
```

```
        try
        {
            BufferedReader br =
                        new BufferedReader(
                                    new FileReader(source));
            PrintWriter pw =
                        new PrintWriter(new FileWriter(target));
            String line = br.readLine();
            while (line != null)
            {
                pw.println(line);
                line= br.readLine();
            }
            br.close();
            pw.close();    // flushes output stream
        }
        catch (IOException e)
        { System.out.println(e); }
        System.out.println("Done");
    }
}
```

Important

Whenever doing output to an OutputStream or a Writer, ensure that the data does not get stuck in an operating-system buffer by calling *flush* in one way or another.

The PrintStream *System.out* automatically flushes its output.

 Copyright 2005 by Ken Slonneger

Two Basic Kinds of Files

1. A text file consists of a sequence of ascii characters.
   - Keyboard produces text.
   - Monitor display consumes text.
2. A binary file consists of bytes representing various types of data such as **int**, **double**, etc.

Example

Text in a file

S	A	I	L	B	O	A	T
53	41	49	4C	42	4F	41	54

As bits

01010011 01000001 01001001 01001100
01000010 01001111 01000001 01010100

Binary (same eight bytes)

Four short:	21313	18764	16975	16724
Two int:	1396787523	1112490324		
One long:	5991567705130437960			
Two float:	8.30158e+11	51.8138		
One double:	1.12681e+93			

File of Integers

- Create a (binary) file of integers.
- Read the file.

Use the data stream classes, subclasses of InputStream and OutputStream, which were designed for binary data.

The classes DataInputStream and DataOutputStream contain methods for handling all of the basic types in Java.

byte readByte()	**void** writeByte(**int** b)
short readShort()	**void** writeShort(**int** s)
int readInt()	**void** writeInt(**int** n)
long readLong()	**void** writeLong(**long** g)
float readFloat()	**void** writeFloat(**float** f)
double readDouble()	**void** writeDouble(**double** d)
char readChar()	**void** writeChar(**int** c)
boolean readBoolean()	**void** writeBoolean(**boolean** b)
String readUTF()	**void** writeUTF(String s)

UTF = Unicode Transformation Format (UTF-8)

Ascii → one byte

Other Unicode → two or three byte sequences

 Copyright 2005 by Ken Slonneger

Create a Binary File of Integers

Name the file “Integers”.

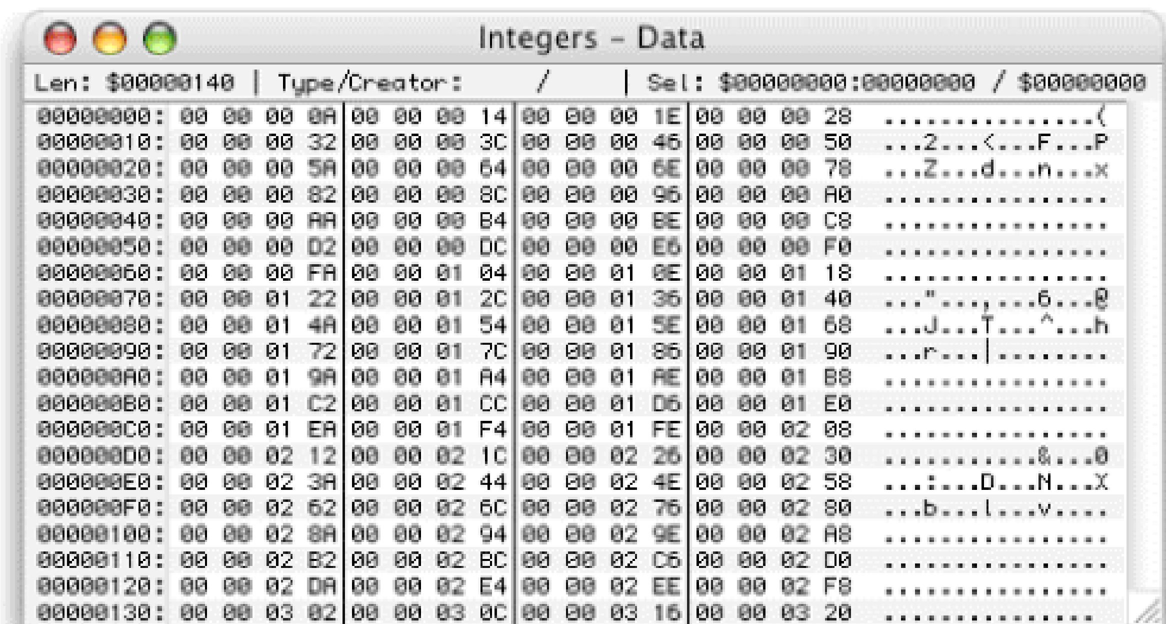
```
import java.io.*;

public class IntFile
{
    public static void main(String [] args)
    {
        try             // Create a file of integers
        {
            DataOutputStream ostream =
                new DataOutputStream(
                    new FileOutputStream("Integers"));

            for (int k=10; k<=800; k=k+10)
                ostream.writeInt(k);

            ostream.close();
        }
        catch (IOException e)
        {   System.out.println(e);
            return;
        }

//-----------------------------------------------------------
```

```
        DataInputStream istream = null;
        try                      // Read the file of integers
        {
            istream = new DataInputStream(
                              new FileInputStream("Integers"));
            while (true)
            {
                int num = istream.readInt();
                System.out.print( num + " " );
            }
        }
        catch (EOFException e)
        { System.out.println("\nEOF reached"); }
        catch (IOException e)
        { System.out.println(e); }
        finally
        {   try { istream.close(); }
            catch (IOException e) { }
        }
    }
}
```

Output

```
10 20 30 40 50 60 70 80 90 100 110 120 130 140 150
160 170 180 190 200 210 220 230 240 250 260 270 280
290 300 310 320 330 340 350 360 370 380 390 400 410
420 430 440 450 460 470 480 490 500 510 520 530 540
550 560 570 580 590 600 610 620 630 640 650 660 670
680 690 700 710 720 730 740 750 760 770 780 790 800
EOF reached
```

 Copyright 2005 by Ken Slonneger



A File of Strings

Use the DataStream methods

String readUTF()

void writeUTF(String str)

UTF (Unicode Transformation Format) refers to the UTF-8 format, an ascii-compatible encoding of Unicode characters.

```
import java.io.*;
public class StringFile
{
    public static void main(String [] args)
    {
        try              // Create a file of Strings
        {
            String s = "Parts of this string are used to "
                                + "provide different lengths";
```

```
            DataOutputStream ostream =
                        new DataOutputStream(
                                new FileOutputStream("Strings"));
            for (int k=1; k<=20; k++)
                ostream.writeUTF("String " + k + ": "
                                     + s.substring(0, 2*k+1));
            ostream.close();
        }
        catch (IOException e)
        {   System.out.println(e);   return;  }

//-------------------------------------------------------------------

        try                     // Read the file of Strings
        {
            DataInputStream istream =
                        new DataInputStream(
                                new FileInputStream("Strings"));
            while (istream.available() > 0)
            {
                String str = istream.readUTF();
                System.out.println(str);
            }
            istream.close();
        }
        catch (IOException e)
        {   System.out.println(e);   }
    }
}
```

Output

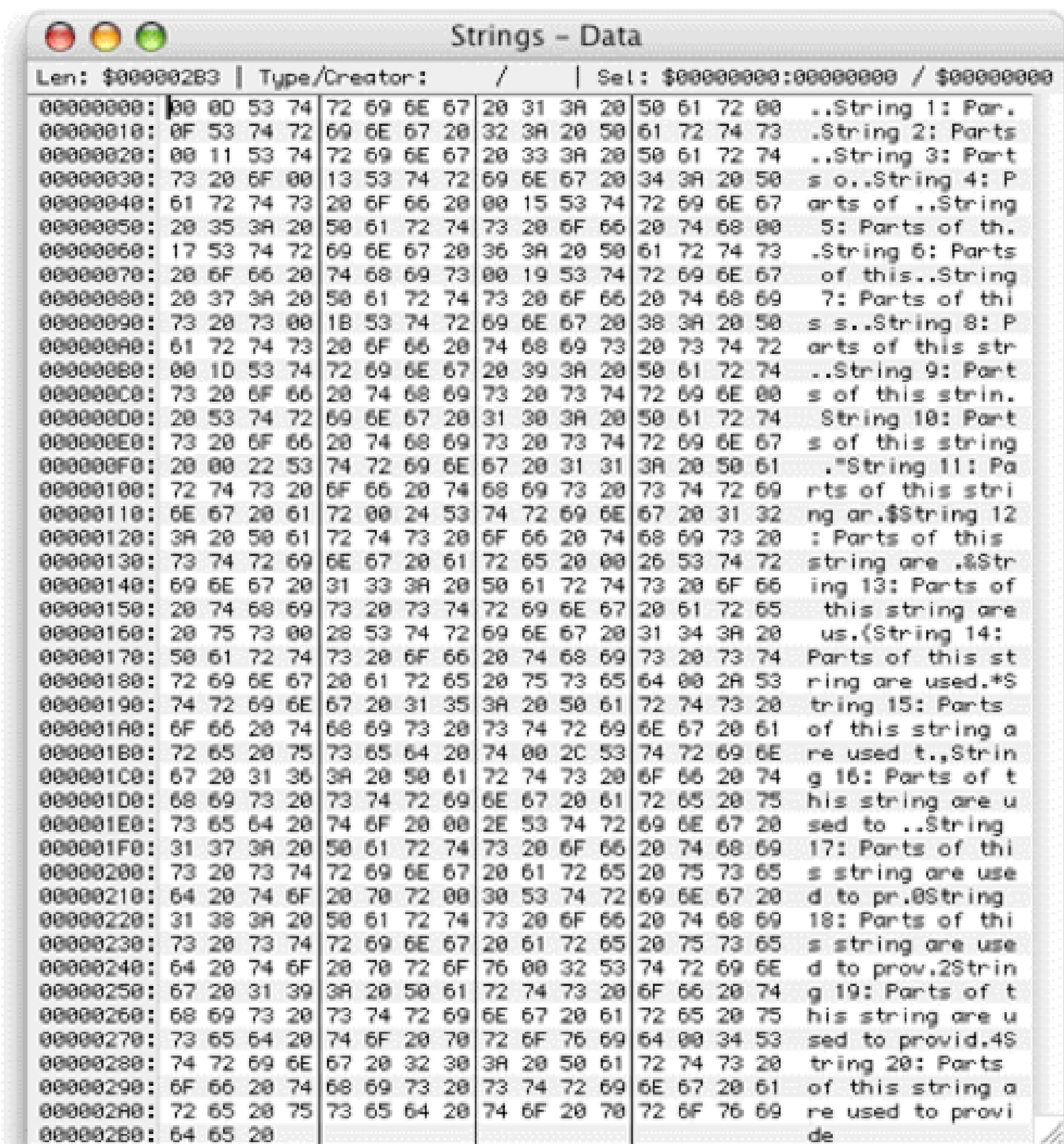
```
String 1: Par
String 2: Parts
String 3: Parts o
```

String 4: Parts of

  :   :

String 19: Parts of this string are used to provid

String 20: Parts of this string are used to provide



Reading Numbers from a Textfile or Keyboard

Three approaches

1. Use a StreamTokenizer object to grab numeric tokens from a Reader.

   Create a StreamTokenizer object:

   ```
   StreamTokenizer stk =
       new StreamTokenizer(new FileReader(fname));
   ```

   Get the code of the next token:

   ```
   int code = stk.nextToken();
   ```

   Compare token code with predefined constants:

   ```
   code == StreamTokenizer.TT_EOF
   code == StreamTokenizer.TT_NUMBER
   code == StreamTokenizer.TT_WORD
   ```

   If the token is a number, get value as a **double** from the instance variable *stk.nval*.

   If the token is a word, get value as a String from the instance variable *stk.sval*.

   If the token is a single character, its integer value is its ascii code.

   If reading integers, cast *stk.nval* to **int**.

 Copyright 2005 by Ken Slonneger

Example

Read a text file and isolate the numbers, words, and individual characters in the file.

Use the default definition of token delimiter in StreamTokenizer.

The specification of the delimiters can be changed if needed.

```
import java.io.*;                       // ScanFile.java

public class ScanFile
{
    public static void main(String [] args)
    {
        PrintStream ps = System.out;          // an abbreviation
        String filename = "";
        if (args.length == 1)
            filename = args[0];
        else
        {   ps.println("Usage: java ScanFile filename");
            return;
        }
        try
        {   FileReader fr = new FileReader(filename);
            StreamTokenizer stk = new StreamTokenizer(fr);

            int code = stk.nextToken();
            while (code!= StreamTokenizer.TT_EOF)
            {
                switch (code)
                {
                    case StreamTokenizer.TT_NUMBER:
                        ps.println("Number: " + stk.nval);
                        break;

                    case StreamTokenizer.TT_WORD:
                        ps.println("String: " + stk.sval);
                        break;
```

```
                    default:
                        ps.println("Character: " + (char)code);
                        break;
                }
                token = stk.nextToken();
            }
        }
        catch (FileNotFoundException fnfe)
        {
            ps.println("File " + filename + " not found");
        }
        catch (IOException e)
        {
            ps.println("Some other IO error");
        }
    }
}
```

Test File

```
This file contains numbers as text.
(34, 78, 12, 7.5, 6.8)
(6.3, 75, 22, 3.9, 11)
Done.
```

Execution Results

```
% java ScanFile test
String: This            Character: ,       Number: 75.0
String: file            Number: 12.0       Character: ,
String: contains        Character: ,       Number: 22.0
String: numbers         Number: 7.5        Character: ,
String: as              Character: ,       Number: 3.9
String: text.           Number: 6.8        Character: ,
Character: (            Character: )       Number: 11.0
Number: 34.0            Character: (       Character: )
Character: ,            Number: 6.3        String: Done.
Number: 78.0            Character: ,
```

 Copyright 2005 by Ken Slonneger

2. Have only one number per line.

Use *readLine* with a BufferedReader object to get a String of digits with possibly a decimal point and/or minus sign

Trim the String to remove extra spaces.

```
String s = br.readLine().trim();
```

Convert to **int** using one of these expressions:

```
int k = Integer.parseInt(s);
int m = new Integer(s).intValue();
int n = Integer.valueOf(s).intValue();
```

Convert to **double** using one of these expressions:

```
double d = Double.parseDouble(s);
double e = new Double(s).doubleValue();
double f = Double.valueOf(s).doubleValue();
```

3. Allow more than one number per line.

- Use *readLine* with a BufferedReader object to get a String *str* containing multiple numbers.
- Use a StringTokenizer object *strTok* to grab tokens, which are always strings, from the String *str* in a manner similar to StreamTokenizer.
- StringTokenizer is found in the package *java.util*.

```
StringTokenizer strTok = new StringTokenizer(str);
String token = strTok.nextToken();
strTok.hasMoreTokens()   returns a boolean value
```

Example: Isolate Tokens from Input Stream

```
BufferedReader br = new BufferedReader(
                              new InputStreamReader(System.in);
try
{   str = br.readLine();              // priming read
    while (!str.equals(""))
    {
        StringTokenizer strTok = new StringTokenizer(str);
        while (strTok.hasMoreTokens())
        {
            String token = strTok.nextToken();
            //   Convert to numeric
            System.out.println(token);
        }
        str = br.readLine();
    }
}
catch (IOException e)  {  }
```

Sample Execution

```
Enter numbers terminated by empty line
123 456
123
456
22.33 55.66 88.999
22.33
55.66
88.999
19 92 38.47
19
92
38.47
```

 Copyright 2005 by Ken Slonneger

Scanner Class

Java 1.5 has added a very useful class for input.

The Scanner class can be found in the *java.util* package.

Constructors

Of the many constructors provided for Scanner, three will be of the most use.

```
Scanner(Reader r)

Scanner(InputStream is)

Scanner(File f)
```

A File object can be created using the code

```
File f = new File("fileName");
```

Using Scanner

A Scanner object recognizes instance methods that return the next token in the input stream.

A token is a chunk of characters with some meaning defined by the characters that make up the token and the characters that delimit the token.

"Word" Tokens

The default delimiters for a Scanner object are the white space characters.

To isolate the "words" in a text file named "textfile", use the following code.

```
Scanner sc = new Scanner(new FileReader("textfile"));
while (sc.hasNext())
{
    String word = sc.next();
    System.out.println(word);
}
```

The Scanner methods do not throw any checked exceptions.

The Scanner class has ways to change the definition of the delimiters for a particular Scanner object.

Reading Primitive Values

The Scanner class has instance methods for reading each of the Java primitive types except **char**.

We consider several examples.

To read a text stream of **int** values:

```
while (sc.hasNextInt())
{
    int m = sc.nextInt();
    System.out.println(m);
}
```

 Copyright 2005 by Ken Slonneger

To read a text stream of **double** values:

```
while (sc.hasNextDouble())
{
      double d = sc.nextDouble();
      System.out.println(d);
}
```

To read a text stream of **boolean** values:

```
while (sc.hasNextBoolean())
{
      boolean b = sc.nextBoolean();
      System.out.println(b);
}
```

Reading Lines

The Scanner class has methods that provide the behavior of the *readLine* method from BufferedReader.

To read lines from a text stream.

```
while (sc.hasNextLine())
{
      String str = sc.nextLine();
      System.out.println(str);
}
```

Closing a Scanner

We can release the resources provided to Scanner with the command:

```
sc.close();
```

Problem

Read a set of numbers from the keyboard and find their sum.

Use zero as a sentinel at the end of the input.

Since the numbers are not specified in more detail, we will read **double** values.

Code: Sum.java

```
import java.util.Scanner;

public class Sum
{
   public static void main(String [] args)
   {
      System.out.println("Enter numbers terminated by a zero.");
      Scanner sc = new Scanner(System.in);
      double sum = 0.0;
      while (true)
      {
         double d = sc.nextDouble();
         if (d==0.0) break;
         sum = sum + d;
      }
      System.out.println("sum = " + sum);
   }
}
```

Sample Execution

```
% java Sum
Enter numbers terminated by a zero.
45.6 23.8 -44.22 12 67.88
20.08 -66.84 586
0
sum = 644.3
```

 Copyright 2005 by Ken Slonneger

HexDump

Program displays the bytes in a file as 16 unsigned hex bytes per line. One byte = two hex digits.

Use a FileInputStream (byte-oriented).

Bytes are coerced to **int** and converted to hex using a class method in Integer.

As formatting, print a zero before each single digit hex byte, since leading zeros are not provided.

```
import java.io.*;
public class HexDump
{
    public static void main(String [] args)
    {
        try
        {
            BufferedReader br =
                        new BufferedReader(
                                    new InputStreamReader(System.in));
            System.out.print("Enter a file name: ");
            String fName = br.readLine().trim();
            FileInputStream istream =
                                    new FileInputStream(fName);
```

```
        byte [] bytes = new byte[16];
        int count = istream.read(bytes);
        while (count != -1)                    // returns -1 at end of file
        {
            for (int k=0; k<count; k++)
            {
                int n = bytes[k];              // Alternative: Replace by
                if (n<0) n = n + 256;          //    int n = bytes[k] & 0xFF;
                String hs = Integer.toHexString(n);
                if (n<16) System.out.print("0");
                System.out.print(hs + " ");
            }
            System.out.println();
            count = istream.read(bytes);
        }
        istream.close();
    }
    catch (IOException e)
    {   System.out.println(e);   }
  }
}
```

Notes

- Method *toHexString* takes an **int** as its parameter.
- If the **int** is negative, *toHexString* returns 8 hex digits.

  Solution: Convert negative byte (-128≤b≤-1) into a positive integer (128≤n≤255) by adding 256.

 Copyright 2005 by Ken Slonneger

Enter a source file name: **Integers**

```
00 00 00 0a 00 00 00 14 00 00 00 1e 00 00 00 28
00 00 00 32 00 00 00 3c 00 00 00 46 00 00 00 50
00 00 00 5a 00 00 00 64 00 00 00 6e 00 00 00 78
00 00 00 82 00 00 00 8c 00 00 00 96 00 00 00 a0
00 00 00 aa 00 00 00 b4 00 00 00 be 00 00 00 c8
00 00 00 d2 00 00 00 dc 00 00 00 e6 00 00 00 f0
00 00 00 fa 00 00 01 04 00 00 01 0e 00 00 01 18
00 00 01 22 00 00 01 2c 00 00 01 36 00 00 01 40
00 00 01 4a 00 00 01 54 00 00 01 5e 00 00 01 68
00 00 01 72 00 00 01 7c 00 00 01 86 00 00 01 90
00 00 01 9a 00 00 01 a4 00 00 01 ae 00 00 01 b8
00 00 01 c2 00 00 01 cc 00 00 01 d6 00 00 01 e0
00 00 01 ea 00 00 01 f4 00 00 01 fe 00 00 02 08
00 00 02 12 00 00 02 1c 00 00 02 26 00 00 02 30
00 00 02 3a 00 00 02 44 00 00 02 4e 00 00 02 58
00 00 02 62 00 00 02 6c 00 00 02 76 00 00 02 80
00 00 02 8a 00 00 02 94 00 00 02 9e 00 00 02 a8
00 00 02 b2 00 00 02 bc 00 00 02 c6 00 00 02 d0
00 00 02 da 00 00 02 e4 00 00 02 ee 00 00 02 f8
00 00 03 02 00 00 03 0c 00 00 03 16 00 00 03 20
```

A Linux/Unix tool

% **xxd Integers**

Serialization

A Java object normally expires when the program that created it terminates. Since no variable refers to it, the garbage collector reclaims its storage.

Problem: Want to make an object be persistant so that it can be saved between program executions.

Possible Solution

If all of the instance variables (fields) in the object are of primitive types or Strings, use

- Methods from DataOutputStream (*writeInt*, *writeDouble*, *writeUTF*, etc.) to store the object.

- Methods from DataInputStream (*readInt*, *readDouble*, *readUTF*, etc.) to restore the object.

Difficulties

1. What if objects contain arrays of varying sizes?

2. What if instance variables are references to other objects, which have references to still other objects, and so on? Imagine a graph of objects that lead from the object to be saved. The entire graph must be saved and restored.

We need a byte-coded representation of objects that can be stored in a file external to Java programs, so that the file can be read later and the objects can be reconstructed.

 Copyright 2005 by Ken Slonneger

Serialization

Serializing an object means to code it as an ordered series of bytes in such a way that it can be rebuilt (really a copy) from the byte stream.

Serialization needs to store enough information so that the original object can be rebuilt, including all objects to which it refers (the object graph).

Java has classes (in the *java.io* package) that allow the creation of streams for object serialization and methods that write to and read from these streams.

Only an object of a class that implements the empty interface java.io.Serializable or a subclass of such a class can be serialized. Such an interface is called a *marker* interface.

What is Saved

- Class of the object.
- Class signature of the object (types of instance variables and signatures of instance methods).
- All instance variables not declared **transient**.
- Obects referred to by non-transient instance variables.

Uses of Serialization

- Make objects persistant.
- Communicate objects over a network.
- Make a copy of an object.

Saving an Object (an array of String)

1. Open a file and create an ObjectOutputStream object.

```
ObjectOutputStream oos =
            new ObjectOutputStream(
                  new FileOutputStream("datafile"));
```

2. Write an object to the stream using writeObject().

```
String [] sa = new String [150];
      // Fill array with 150 strings, say names of students
oos.writeObject(sa);     // Save object (the array)
oos.flush();             // Empty output buffer
```

Restoring the Object

1. Open the file and create an ObjectInputStream object.

```
ObjectInputStream ois =
            new ObjectInputStream(
                  new FileInputStream("datafile"));
```

2. Read the object from the stream using readObject() and then cast it to its appropriate type.

```
String [] newSa;
      // Restore the object (readObject returns an Object)
newSa = (String [])ois.readObject();

      // May throw checked ClassNotFoundException.
```

When an object is retrieved from a stream, it is validated to ensure that it can be rebuilt as the intended object.

Constructors and operations may throw various IOExceptions.

 Copyright 2005 by Ken Slonneger

Conditions

- A class whose objects are to be saved must implement the interface Serializable, a *marker* interface.
- The class must be visible at the point of serialization.

The *implements Serializable* clause acts as a tag indicating the possibility of serializing the objects of the class.

Primitive Data

ObjectOutputStream and ObjectInputStream also implement methods for writing and reading primitive data and Strings from the interfaces DataOutput and DataInput:

writeChar	readChar	
writeInt	readInt	
writeDouble	readDouble	
writeUTF	readUTF	and so on.

Some Classes that Implement Serializable

String	StringBuffer	Calendar	Date
ArrayList	Character	Boolean	Number
LinkedList	Component	Color	Font
Point	Throwable	InetAddress	URL

Note: No methods or class variables are saved when an object is serialized. A class knows which methods and static data are defined in it.

Save a Domino set

Create a complete set of dominoes in an array and store it in a file named “doms”.

```
import java.io.*;
class Domino implements Serializable
{
// Instance Variables
    private int spots1, spots2;
    private boolean faceUp;

// Class Variables
    static final int MAXSPOTS = 9;
    static int numDominoes=0;

// Constructors
    Domino(int val1, int val2, boolean up)
    {
        if (0<=val1 && val1<=MAXSPOTS)      // validation
            spots1 = val1;
        else spots1 = 0;
        if (0<=val2 && val2<=MAXSPOTS)   spots2 = val2;
        else spots2 = 0;
        faceUp = up;
        numDominoes++;
    }

    Domino(boolean up)                    // a random domino
    {
        spots1 = (int)((MAXSPOTS + 1) * Math.random());
        spots2 = (int)((MAXSPOTS + 1) * Math.random());
        faceUp = up;
        numDominoes++;
    }
```

 Copyright 2005 by Ken Slonneger

```
    Domino()                                // a default domino
    {     this(0, 0, false);      }

// Instance Methods
    int getHigh()                           // an accessor
    {
          if (spots1>= spots2) return spots1;
          else return spots2;
    }

    int getLow()                            // an accessor
    {
          if (spots1<= spots2) return spots1;
          else return spots2;
    }

    public String toString()
     {
           String orientation = "DOWN";
           if (faceUp)  orientation = "UP";
           return "<" + getLow() + ", " + getHigh() + ">  " + orientation;
     }

    boolean matches(Domino otherDomino)
    {
        int a = otherDomino.getHigh();
        int b = otherDomino.getLow();
        int x = getHigh();
        int y = getLow();
        return a==x || a==y || b==x || b==y;
    }

// Class Methods
    static int getNumber()
    {   return numDominoes;       }
}
```

```
public class SaveDoms
{
    static int size =
            (Domino.MAXSPOTS+1)*(Domino.MAXSPOTS+2) / 2;

    public static void main(String [] args)
    {
        Domino [] dominoSet = new Domino [size];
        int index = 0;
        for (int m=0; m<=Domino.MAXSPOTS; m++)
            for (int n=m; n<=Domino.MAXSPOTS; n++)
            {
                dominoSet[index] = new Domino(m, n, false);
                index++;
            }

        try
        {
            ObjectOutputStream save =
                    new ObjectOutputStream(
                        new FileOutputStream("doms"));

            save.writeObject(dominoSet);

            save.flush();
            save.close();
        }
        catch (IOException e)
        { System.out.println(e); }
    }
}
```

 Copyright 2005 by Ken Slonneger

Restore Domino Set

```
import java.io.*;
public class RestoreDoms
{
    public static void main(String [] args)
    {
        Domino [] dominoSet = null;
        try
        {
            ObjectInputStream restore =
                    new ObjectInputStream(
                        new FileInputStream("doms"));
            dominoSet = (Domino [])restore.readObject();
            restore.close();
        }
        catch (IOException e)
        {  System.out.println(e);  }
        catch (ClassNotFoundException e)
        {  System.out.println(e);  }
        for (int k=0; k<dominoSet.length; k++)
            System.out.println(dominoSet[k]);
    }
}

class Domino implements Serializable
{
    private int spots1, spots2;
          :
}
```

Output

<0, 0> DOWN	<2, 2> DOWN	<4, 7> DOWN
<0, 1> DOWN	<2, 3> DOWN	<4, 8> DOWN
<0, 2> DOWN	<2, 4> DOWN	<4, 9> DOWN
<0, 3> DOWN	<2, 5> DOWN	<5, 5> DOWN
<0, 4> DOWN	<2, 6> DOWN	<5, 6> DOWN
<0, 5> DOWN	<2, 7> DOWN	<5, 7> DOWN
<0, 6> DOWN	<2, 8> DOWN	<5, 8> DOWN
<0, 7> DOWN	<2, 9> DOWN	<5, 9> DOWN
<0, 8> DOWN	<3, 3> DOWN	<6, 6> DOWN
<0, 9> DOWN	<3, 4> DOWN	<6, 7> DOWN
<1, 1> DOWN	<3, 5> DOWN	<6, 8> DOWN
<1, 2> DOWN	<3, 6> DOWN	<6, 9> DOWN
<1, 3> DOWN	<3, 7> DOWN	<7, 7> DOWN
<1, 4> DOWN	<3, 8> DOWN	<7, 8> DOWN
<1, 5> DOWN	<3, 9> DOWN	<7, 9> DOWN
<1, 6> DOWN	<4, 4> DOWN	<8, 8> DOWN
<1, 7> DOWN	<4, 5> DOWN	<8, 9> DOWN
<1, 8> DOWN	<4, 6> DOWN	<9, 9> DOWN
<1, 9> DOWN		

 Copyright 2005 by Ken Slonneger