

COP 3330: Object-Oriented Programming Summer 2007

Java I/O

Instructor : Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 823-2790
<http://www.cs.ucf.edu/courses/cop3330/sum2007>

School of Electrical Engineering and Computer Science
University of Central Florida



Reading and Writing in Java

- The vast majority of computer programs require information to be entered into them, and programs usually provide some form of output information.
- Early in this course you created output using the simple `print()` and `println()` methods of the `java.lang.System` class and more recently you've graduated to using the `JOptionPane` class methods to display information to the user in a GUI.
- Many programs rely on input information being contained in a data file that the program must read and process. In turn, the program writes information to an output file.



Reading and Writing in Java (cont.)

- Java provides many classes to perform the program input and output.
- **Program I/O** refers to any I/O performed by the program.
- **File I/O** refers specifically to I/O performed on files.
- Program I/O can come from or be sent to the monitor screen, data files, network sockets, or the keyboard.



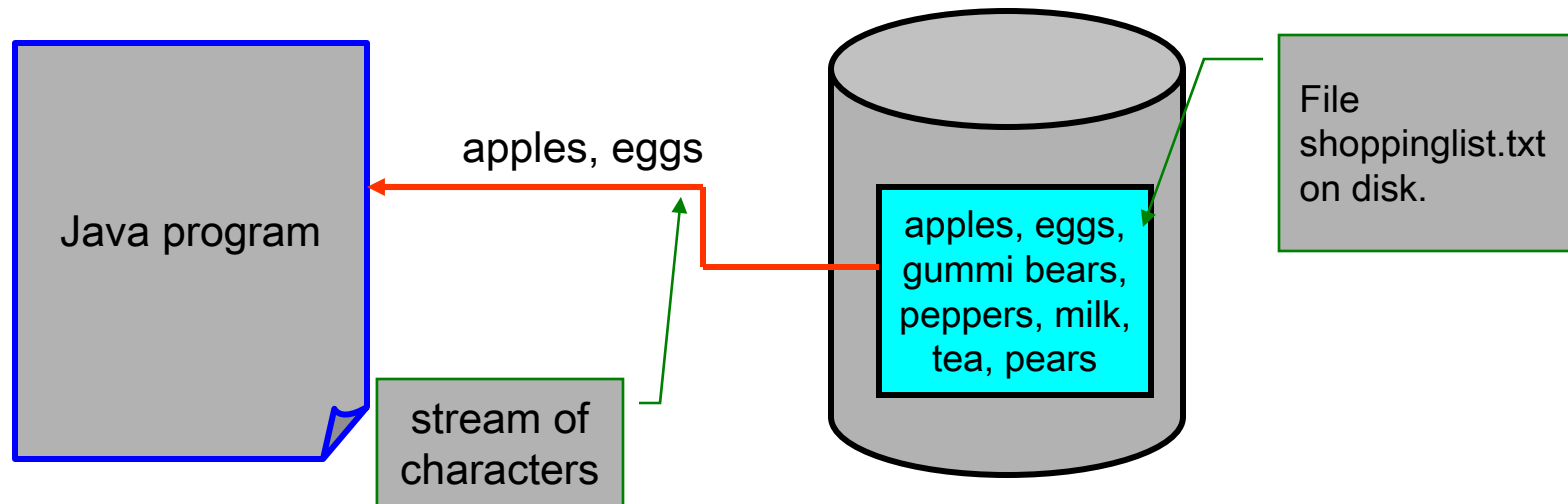
Reading and Writing in Java (cont.)

- Java designers constructed program I/O to be based on three principles:
 1. The input and output is based on **streams** that have a physical device at one end, such as a disk file, and data streams into or out of the program in a flow of characters or bytes. Classes are used to manage how the data comes into or leaves a program.
 2. I/O should be portable and should obtain consistent results even though the platforms may differ.
 3. Java provides many classes that each perform a few tasks instead of large classes that do many things.



Streams

- A way to visualize data flowing into or out of a Java program is to envision a stream of characters or a data pipeline.
- This stream of data is linked to a physical device, such as a file stored on the hard drive to a network socket.



Streams (cont.)

- There are two types of streams in Java, **byte streams** and **character streams**.
- **Byte streams** pump the data into and out of programs as bytes. Byte stream classes are used for handling these bytes. Binary data are stored either as 8-bit bytes or as an ASCII character code set.
- **Character streams** are used for handling character data. Character streams use Unicode, which is composed of two-byte characters and can be used on an international basis.



I/O with Byte Stream Classes

- Byte stream classes in Java use a two-class hierarchical structure, one for reading and one for writing.
- The top two classes are `InputStream` and `OutputStream`.
- Each of these superclasses has many subclasses designed for working with different devices such as files and network connections.
- Important methods include `read()`, and `write()`.
- For reading and writing data files, use the `FileInputStream` and `FileOutputStream` class objects. The simplest form of these classes reads and writes data one byte at a time.



I/O with Byte Stream Classes (cont.)

- `FileInputStream` reads data, one byte at a time, and treats the data as integers.
- If the `FileInputStream` object tries to read beyond the last character in the file, the `read` method returns a negative one (-1).



Byte Stream Methods

- The `FileInputStream` class has several `read()` methods as well as other supporting methods. Some of these methods are listed below.

`int available():` Returns the number of bytes available that can be read in this file.

`int read():` Reads one byte at a time and returns each byte as an integer.

`void read(byte[] b):` Reads up to `b.length` bytes of data from the input stream.

`void read (byte[] b, int offset, int length):` Reads length number of bytes from the input stream beginning at the offset of the data.

`void close():` Closes the file and releases resources that are associated with the input stream.



Byte Stream Methods (cont.)

- The `FileOutputStream` class has several `write()` methods as well as other supporting methods. Some of these methods are listed below.

`void write(int b):` Writes the byte to the output stream. The input is a single integer and is converted to a byte.

`void write(byte[] b):` Writes up to `b.length` bytes of data into the output file stream.

`void write (byte[] b, int from, int length):` Writes a portion of the byte array to the output file stream. The *from* variable indicates the starting index, and the *length* is the number of bytes.

`void close():` Closes the file and releases any resources that are associated with the output stream.




Technique 1: Reading a File – 1 byte at a Time

- The first example program in this section of notes demonstrates how to read a text data file one byte at a time, and print each byte read as a character.
- The text file will be called: `shoppinglist.txt` and contains the following:

```
apples, eggs, gummi bears  
peppers, milk, coke  
tea, pears, chicken  
shrimp, onions, frosted flakes
```

shoppinglist.txt



ReadFile1.java

```
//File:  ReadFile1.java
// This program reads a file using FileInputStream object.
// It reads the file a byte at a time and prints the char
// to the screen.
import java.io.FileInputStream;
import java.io.IOException;
import java.io.FileNotFoundException;
public class ReadFile1
{
    public static void main( String[] args)
    {
        ReadFile1 app = new ReadFile1();
        System.exit(0);
    }
    public ReadFile1()
    {
        //We create a FileInputStream object and pass the name of the
        //data file into the constructor. If Java can't find the file,
        //it throws a FileNotFoundException.
        try
        {
```



```

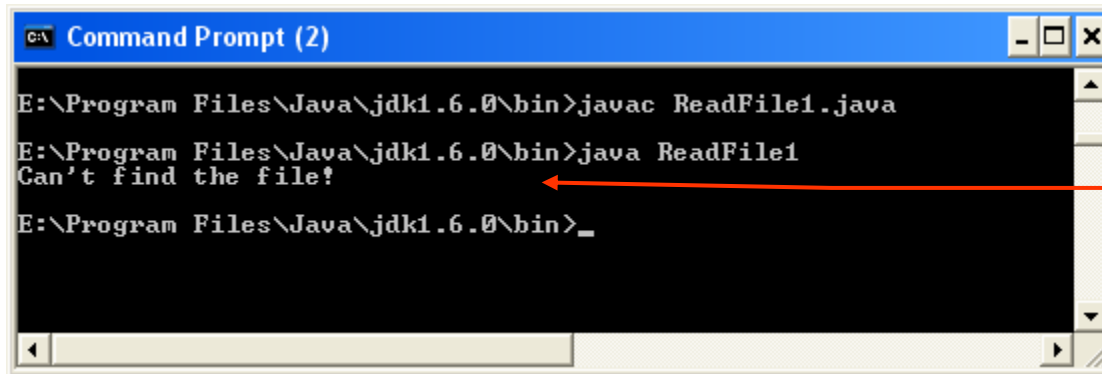
        FileInputStream fileIn = new FileInputStream
            ("shoppinglist.txt");
        //Ask the file object how many bytes are in the file.
        int size = fileIn.available();
        int oneChar;
        for(int i = 0; i < size; ++i){
            //Read the file one byte at a time.
            //If a read error then throw an IOException.
            oneChar = fileIn.read();

            //print without linefeeds
            System.out.print((char)oneChar);
        }
        fileIn.close();
    }
    catch(FileNotFoundException fnfe){
        System.out.println("Can't find the file!");
    }
    catch(IOException ioe){
        System.out.println("Problem reading the file!");
    }
}
}

```



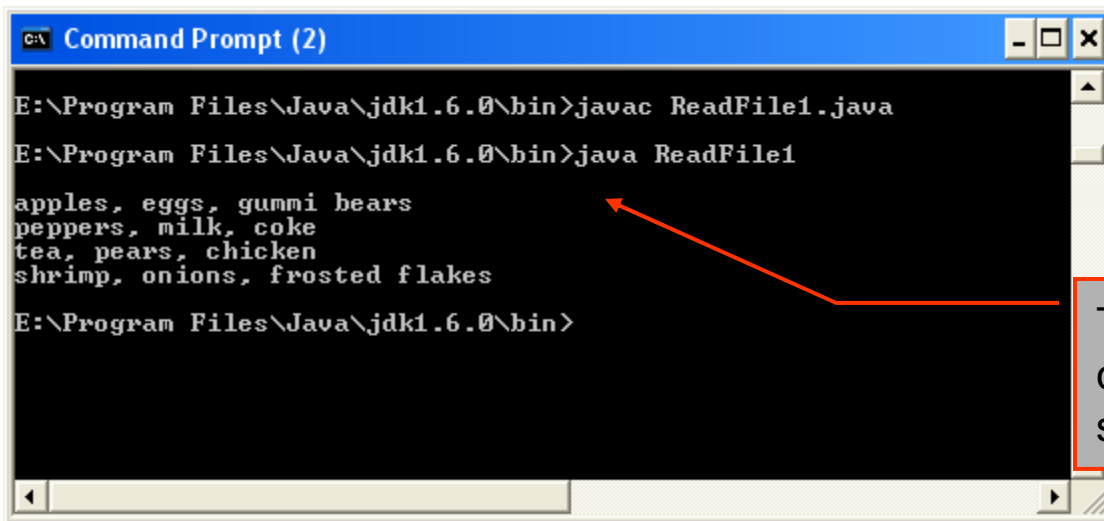
Output from ReadFile1.java



```
C:\ Command Prompt (2)

E:\Program Files\Java\jdk1.6.0\bin>javac ReadFile1.java
E:\Program Files\Java\jdk1.6.0\bin>java ReadFile1
Can't find the file!
E:\Program Files\Java\jdk1.6.0\bin>_
```

This execution specified the file named "grocerylist.txt", which does not exist.



```
C:\ Command Prompt (2)

E:\Program Files\Java\jdk1.6.0\bin>javac ReadFile1.java
E:\Program Files\Java\jdk1.6.0\bin>java ReadFile1
apples, eggs, gummi bears
peppers, milk, coke
tea, pears, chicken
shrimp, onions, frosted flakes
E:\Program Files\Java\jdk1.6.0\bin>
```

This execution specified the correct file which was successfully opened and printed.



Explanation of ReadFile1.java

- Once the `FileInputStream` object is created, the object is queried to determine its size. The `available()` method returns the number of bytes that can be read from this file.
- In the for loop, one byte at a time is read and printed to the command prompt window.
 - Notice that `print()` was used which does not add a new line character. The new line characters are already in the data file. When the “\n” characters are read and output via the `print()` method, we’ll see the linefeed in the output.



Explanation of ReadFile1.java (cont.)

- There is a second manner in which this code could be written which involves checking each integer as it is read in, to see if it is a negative one (-1).
- Using this technique, you do not use the `available()` method to determine the number of bytes in the file, but rather just read until the last byte is read.
- This modification is shown in the next version of the `ReadFile1.java` program. Try this modification yourself to see that it produces exactly the same results.



Slightly Modified ReadFile1.java

```
//File:  ReadFile1.java
// This program reads a file using FileInputStream object.
// It reads the file a byte at a time and prints the char
// to the screen.
import java.io.FileInputStream;
import java.io.IOException;
import java.io.FileNotFoundException;
public class ReadFile1
{
    public static void main( String[] args)
    {
        ReadFile1 app = new ReadFile1();
        System.exit(0);
    }
    public ReadFile1()
    {
        //We create a FileInputStream object and pass the name of the
        //data file into the constructor. If Java can't find the file,
        //it throws a FileNotFoundException.
        try
        {
```



```

        FileInputStream fileIn = new FileInputStream
            ("shoppinglist.txt");
        //Ask the file object how many bytes are in the file.
        int size = fileIn.available();
        int oneChar;
        //modified way of reading the file - 1 int at a time
        while( ( oneChar = fileIn.read() ) != -1)
        {
            //print without linefeeds
            System.out.print((char)oneChar);
        }
        fileIn.close();
    }
    catch(FileNotFoundException fnfe){
        System.out.println("Can't find the file!");
    }
    catch(IOException ioe){
        System.out.println("Problem reading the file!");
    }
}
}

```



Technique 2: Reading a File into a byte Array

- For our second example using byte streams, we'll read an entire file into a byte array using a single read statement.
- Just to do something to the data that is read, we'll reverse the characters in this array and write the reversed array to an output file.
 - The name of the input file will again be "shoppinglist.txt" and the reversed file that will be created by the executing program will be named "tsilgnippohs.txt" .



ReadFile2.java

```
//File:  ReadFile2.java
// This program reads a file using FileInputStream object.
// It reads the entire file into a byte array in one read statement.
// We then reverse the elements and write it to an output file.
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.FileNotFoundException;
public class ReadFile2
{
    public static void main( String[] args)
    {
        ReadFile2 app = new ReadFile2();
        System.exit(0);
    }
    public ReadFile2()
    {
        //We create FileInputStream and FileOutputStream objects,
        //passing the filename to each constructor.
        try
        {
```



```

        FileInputStream fileIn = new FileInputStream
            ("shoppinglist.txt");
        FileOutputStream fileOut = new FileOutputStream
            ( "tsilgnippohs.txt");
        //Ask the file object how many bytes are in the file.
        int size = fileIn.available();
        byte array[] = new byte[size];
        byte reversedArray[] = new byte[size];
        fileIn.read( array);
        fileIn.close(); //done reading, close the file
        //print the original array
        System.out.println("\n The original array is: \n");
        System.out.print( new String( array));
        for(int i = 0; i < size; ++i){
            reversedArray[i] = array[size - i - 1];
        }
        //print the reversed array
        System.out.println("\n\n The reversed array is: \n ");
        System.out.print( new String( reversedArray));
        fileOut.write( reversedArray );
    }
    catch(FileNotFoundException fnfe){
        System.out.println("Can't find the file!");
    }
    catch(IOException ioe){
        System.out.println("Problem reading the file!");
    }
} } }

```



Output from ReadFile2.java

```
Command Prompt (2)
E:\Program Files\Java\jdk1.6.0\bin>javac ReadFile2.java
E:\Program Files\Java\jdk1.6.0\bin>java ReadFile2
The original array is:
apples, eggs, gummi bears
peppers, milk, coke
tea, pears, chicken
shrimp, onions, frosted flakes
The reversed array is:
sekalf detsofr ,snoino ,pmirhs
nekcihc ,sraep ,aet
ekoc ,klin ,sreppep
sraeh immug ,sgge ,selppa
E:\Program Files\Java\jdk1.6.0\bin>
```

This execution specified the file named "shoppinglist.txt", which was read into an array and the array contents are printed on the screen.

This execution specified the output file as "tsilgnippohs.txt" which holds the reversed contents and will appear in the default directory as a .txt file.



Buffered Character Stream File I/O

- Reading and writing program data using byte stream classes is straightforward, but it presents problems for the Java programmer.
- The data comes into the program as bytes (integers) or in byte arrays. (integer arrays). If the programmer needs to work with each data item in the byte array, they would need to find a way to separate the individual data items.
 - For example, if we needed to list the items in our shopping list, we would need to go through the byte array, pulling out the letters, and starting a new item when we encountered a comma or a new line.
 - If you're thinking that there must be a better way to do this, you're right!



Buffered Character Stream File I/O (cont.)

- Wrapper Classes

- Java provides many wrapper classes. A wrapper class is a programming term that is part of the Java jargon. If you look for Java classes with “Wrapper” in the name, you will not find any.
- Wrapper classes wrap one class in another class, thus improving the features of the first class.
- You’ve already used the `Integer`, `Double`, and `Float` wrapper classes. Each of these classes wraps a primitive single data type value into a class and provides useful methods for the programmer who is working with the primitive values.
- We have used the `BufferedReader` wrapper class extensively in sample code. This wrapper class provides a `readLine()` methods to read data one line at a time.



Buffered Character Stream File I/O (cont.)

- For I/O, Java provide **stream buffering classes** that provide the programmer with a means to attach a memory buffer to the I/O streams.
- Having a memory buffer attached to the I/O stream allows the programmer to work on more than one byte or character at a time.
- There are buffered classes for both byte streams and character stream use.
- We've already wrapped file readers in buffered readers and we'll see how to do the same with file writers wrapped into a buffered writer class.



Buffered Character Stream File I/O (cont.)

- Java's `FileReader` class allows you to read a data file as characters instead of bytes.
- We've wrapped the `FileReader` in a `BufferedReader` class, which provides a `readLine()` method and the ability to read a data file one line at a time.

```
BufferedReader br = new BufferedReader( new FileReader(FILE_NAME));
```



BufferedReader Methods

void close(): Closes the file and releases any resources associated with the output stream.

void mark(int readAheadLimit): Marks the present position in the stream.

boolean markSupported(): Returns true if this stream supports the mark() operation. The BufferedReader class supports this operation.

int read(): Reads a single character.

int read(char[] buf, int offset, int length): Reads characters into a portion of an array.

String readLine(): Reads a line of text.

boolean ready(): Returns true if this stream is ready to be read.

void reset(): Resets the stream to the most recent mark.

long skip(long n): Skips n characters in the stream. Returns the number of characters actually skipped.



Buffered Reader Example: `readStates.java`

- Although we've used the `BufferedReader` class in many examples in class up to this point, I've included a couple more examples here for the sake of completeness and consistency.
- In the first program, to make it a little bit different from some of the others, I've used vectors and made the program a GUI. The second one is also a GUI but reads multiple file lines per activation.
- When programming with data files in any language, it is important that the programmer know how the data file is designed. The programmer must write the corresponding read statements to match the file design in order to read the file accurately.
 - In this case the file consists of one state name per line in the file.



readStates.java

```
//readStates.java
//Use a BufferedReader and FileReader to read a data file one line at a
time.
import java.util.Vector;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.awt.event.WindowEvent;
import java.awt.event.WindowAdapter;
import javax.swing.JFrame;
import javax.swing.JComboBox;

public class readStates extends JFrame
{
    //Create a Vector object, which is a dynamic array that hold objects.
    private static Vector stateList = new Vector();

    public readStates(){
        super( "State List");
    }

    private void readStateList() throws IOException
    {
```



```

// Convenience class for reading character files.
FileReader fr = new FileReader( "StateList.txt");
//Read text from a character-input stream, buffering characters so as
//to provide for the efficient reading of characters, arrays, and
//lines.
BufferedReader br = new BufferedReader( fr );
// Holds the entire line read by BufferedReaders
String line;
//The ready() method returns true as long as there are lines to read.
while( br.ready())
{
    //Use the buffered reader to read the string till \n
    line = br.readLine();

    System.out.println(line); //print line to the command window
    stateList.add( line);    //add each line to the array
}
// close the Buffered Reader
br.close();
}
public static void main( String[] args) {
    readStates app = new readStates();
    try{
        app.readStateList();
    }
}

```



```

        catch( IOException ioe){
            ioe.printStackTrace();
            System.exit( 1);
        }

//add a listener
app.addWindowListener( new WindowAdapter()
{
    public void windowClosing( WindowEvent e)
    {
        System.exit( 0);
    }
});
app.setSize( 200,75);

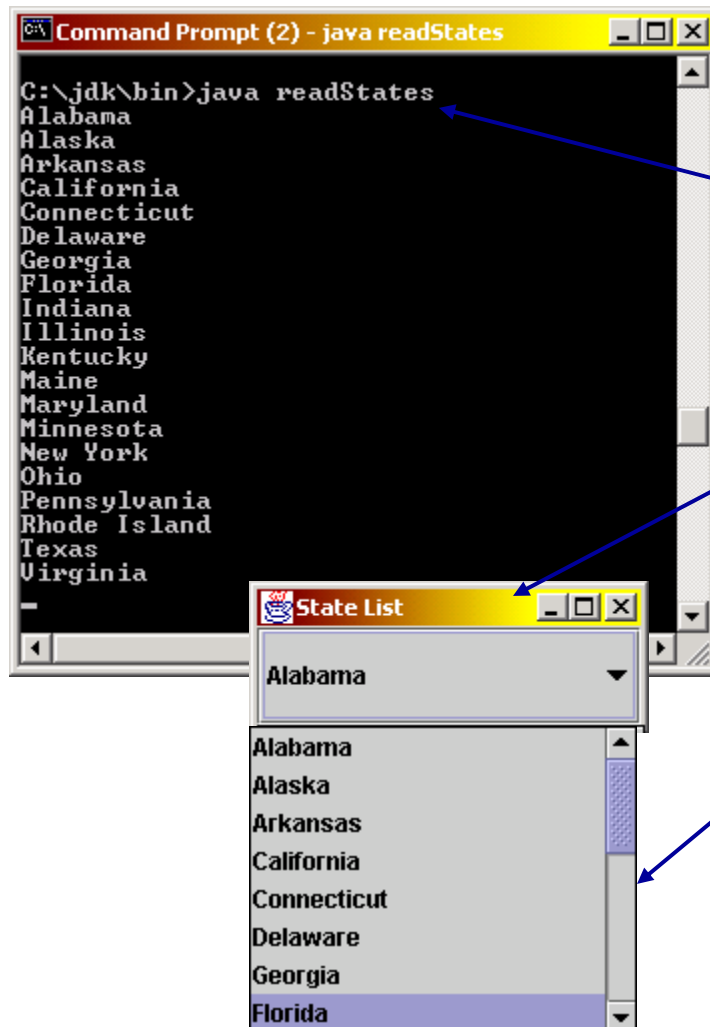
//Get the frame's content pane and add the combo box to it.
app.getContentPane().add( new JComboBox( stateList));
app.show();

    }
}

```



Output from readStates.java



This part of the execution prints the contents of the file to the command prompt window.

This is the GUI before clicking on the state button. The drop-down menu is not shown.

The drop-down menu appears when the GUI button is clicked given the user the option of selecting another state.



Buffered Reader Example 2: `readWeather.java`

- The `BufferedReader readLine()` method is handy anytime the data file is organized with data on individual lines.
- In this next example, the data file represents weather data as a mixture of textual and numeric information, yet the `readLine()` method is used for all the lines. The file is organized in the following manner:

```
Date  
Reporting Station  
High Temperature in Fahrenheit degrees  
Low Temperature in Fahrenheit degrees  
Relative Humidity at 12 noon stated as a percentage 0.xx  
Rainfall total in inches for past 24 hours
```



readWeather.java

```
//readWeather.java
//We read the data on six lines using a separate read statement
//for each piece of data.
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.FileNotFoundException;
import javax.swing.JOptionPane;

public class readWeather {
    public static void main( String[] args) throws IOException
    {
        final String FILENAME = "WeatherSummary.txt";
        int exitCode = 0;
        try{
            BufferedReader br = new BufferedReader(new FileReader(FILENAME));
            //The line holds the line read by BufferedReaders
            String line, output;
            String reportingStation, date;
            double highTemp, lowTemp, humidity, rainfall;
            //We have to make 6 separate read statements to gather the data
            //from the file.
            //The readLine throws IOException if there's a problem.
```



```

        // first line is the date
        date = br.readLine();
        // second line is the station
        reportingStation = br.readLine();
        // third line is the high temp
        line = br.readLine();
        highTemp = Double.parseDouble(line);
        //fourth line is the low temp
        //combine into one line
        lowTemp = Double.parseDouble( br.readLine() );
        // fifth line is the humidity
        humidity = Double.parseDouble( br.readLine() );
        humidity *= 100.0;
        //last line is the rainfall
        rainfall = Double.parseDouble( br.readLine() );
        output = "Date: " + date + "\nStation: " + reportingStation +
            "\nTemp Range: " + highTemp + " to " + lowTemp +
            "\nHumidity at noon: " + humidity + "% \n(Rainfall =
            "+ rainfall + " \")";
        JOptionPane.showMessageDialog(null, output, FILENAME, 1);
        br.close();
    }
    catch(FileNotFoundException fnfe) {
        JOptionPane.showMessageDialog(null, "Can't find the file!",
            FILENAME, 2);
        exitCode = 1; //had a problem
    }
}

```



```
        catch(IOException ioe)    {  
            JOptionPane.showMessageDialog(null, "Trouble!",  
                FILENAME, 2);  
            exitCode = 1;  
        }  
        System.exit( exitCode);  
    }  
}
```



Output from readWeather.java



String Tokenizers

- If the data file is organized so that there is one data item on each line, the job of reading the data from the file is simple. The `BufferedReader`'s `readLine()` method returns each line as a `String` and it can be converted to a numeric value if necessary.
- What happens when there is more than one data item per line in the file? Maybe there is a series of text items or numbers that are separated by commas in the file. There may also be characters in the file that you do not want to process in the program. How do you handle these situations?
- Java has a helper class, called **`StringTokenizer`**, which helps to separate individual parts of the `String` read in by the `readLine()` method.



String Tokenizers (cont.)

- The `StringTokenizer` class is most helpful when the program reads textual data from a file.
- The lines read from the data file are read as `String`s. The `StringTokenizer` can be used on any `String`, such as the input from `JOptionPane.showInputDialog()`, a `String` that you initialize in a program, or a `String` that is filled by a `readLine()` method.
- The `StringTokenizer` class has limited capabilities involving what can be pulled out of the `String`. A good rule of thumb is that if a single character separates data items in the line, such as a comma, or a space, the `StringTokenizer` class is the one to use.



String Tokenizers (cont.)

- For more precise or complicated pattern matching, you would need to use the `Pattern` and `Matcher` classes from Java's **`java.util.regex`** package.
- Recall the example `ReadFile1.java` from page 12 in this set of notes. This program read a “shoppinglist.txt” of 3 items per line which were separated by commas and spaces in the lines of the data file (see page 11). In `ReadFile1.java` we read this file one byte at a time and simply echoed it to the output.
- In the next example, we will read the same file but there will be only commas in this file, no spaces between the data items. If we want to handle each of the items in the list separately, we need to pull each item off the line when the program reads the line from the file.



String Tokenizers (cont.)

- As the first line of the file is read into a String variable, we must pull out “apples”, “eggs”, and “gummi bears” and place them in separate variables. This is the job of the StringTokenizer.
- The StringTokenizer is passed then String we want it to work on, and we must tell it what delimits each data item. In this case we need to tell it that the delimiter in the file is a comma.
- We’ll ask the tokenizer object how many tokens it finds in our line and then we’ll loop through the object, extracting the strings that are delimited by the commas.
- The program `readFileToken.java` illustrates this process.



readFileToken.java

```
//File: readFileToken.java    uses data file shoppinglist.txt
//Use a StringTokenizer to separate data items
//from a String read by the BufferedReader.

import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.FileNotFoundException;
import java.util.StringTokenizer;
import javax.swing.JOptionPane;

public class readFileToken {
    public static void main( String[] args) throws IOException {
        final String FILENAME = "shoppinglist.txt";
        int exitCode = 0;
        try {
            //FileNotFoundException thrown if we can't find the file.
            BufferedReader br = new BufferedReader(new FileReader(FILENAME));
            //The line holds the line read by BufferedReader
            String line;
            // The string tokenizer class allows an application
            // to break a string into tokens.
            StringTokenizer token;
```

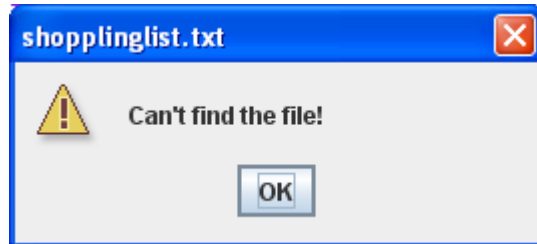


The
StringTokenizer
is operating on
String variable
line and the
delimiter is
specified as a ,.

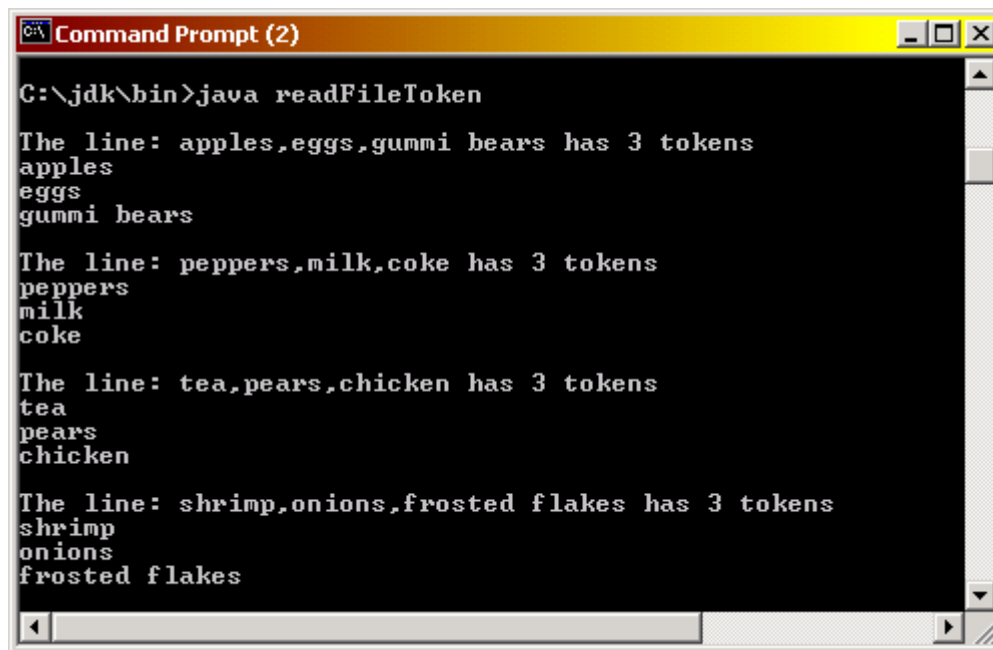
```
//both ready and read might throw IOException
while( br.ready() ) {
    String storeItem;
    // use the buffered reader to read the string till \n
    line = br.readLine();
    // construct with "," as the element delimiter
    token = new StringTokenizer( line, ",");
    int howManyTokens = token.countTokens();
    System.out.println("\nThe line: " + line + " has " +
        howManyTokens + " tokens");
    for(int i = 0; i < howManyTokens; ++i){
        storeItem = token.nextToken();
        System.out.println(storeItem);
    }
}
// close the Buffered Reader
br.close();
}
catch(FileNotFoundException fnfe) {
    JOptionPane.showMessageDialog(null, "Can't find the file!",
        FILENAME, 2);
    exitCode = 1; //had a problem
}
catch(IOException ioe){
    JOptionPane.showMessageDialog(null, "Trouble!", FILENAME, 2);
    exitCode = 1;
}
System.exit( exitCode);
} }
```



Output from readFileToken.java



This execution specified the file named "shoppinglist.txt", which was not a valid file. This generated an error message through the exception handler.



```
C:\jdk\bin>java readFileToken

The line: apples,eggs,gummi bears has 3 tokens
apples
eggs
gummi bears

The line: peppers,milk,coke has 3 tokens
peppers
milk
coke

The line: tea,pears,chicken has 3 tokens
tea
pears
chicken

The line: shrimp,onions,frosted flakes has 3 tokens
shrimp
onions
frosted flakes
```

This execution specified a correct input file. You can see the output from the StringTokenizer



String Tokenizers (cont.)

- In this next example of using the `StringTokenizer` class, everything is basically the same as it was for the `readFileToken.java` program with the exception that now, there are a varying number of items per line in the data file. In addition, we'll add some characters to the file that we want the tokenizer to strip out for us.

hammer, nails, #10 x 1-1/4" wood screws
10d nails, hack saw blades, teflon tape, ruler
pneumatic finishing nailer, pliers
#6 x 1" allen head bolt
PVC cement, PVC primer, scroll saw
7/16" open end wrench, 1/2" impact socket

file: toollist.txt



String Tokenizers (cont.)

- These tasks pose no problem for the `StringTokenizer`. For example, if we are not sure how many tokens appear on a given line of the data file, we'll simply run a loop extracting values and use the `hasMoreTokens()` method, which returns true if there are more tokens in the tokenizer object.
- The format of the file we will use will contain a space after each comma. We want to strip out this leading space so that the data items are correctly represented. Thus, when we extract "ruler" from the third line, the tokenizer extracts " ruler" because it extracts the data between the commas, including the leading space.



String Tokenizers (cont.)

- Use the String class's `trim()` method that returns a copy of the String with leading and trailing whitespace characters omitted.
- Once this is done the strings are stored in a Vector object named `partsList`. [Recall that a vector is a dynamic array (see `java.util.Vector` for more).]



readTools.java

```
//File: readTools.java    uses data file toollist.txt
//Use a StringTokenizer to separate comma delimited data items with leading
//spaces from a String read by the BufferedReader.
//Add them into a Vector to be sorted into alphabetical order.

import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.FileNotFoundException;
import java.util.StringTokenizer;
import javax.swing.JOptionPane;
import java.util.Vector;
import java.util.Collections;

public class readTools {
    public static void main( String[] args) throws IOException {
        final String FILENAME = "PartsList1.txt";
        int exitCode = 0;
        Vector partsList = new Vector();
        try {
            //FileNotFoundException thrown if we can't find the file.
            BufferedReader bufReader = new BufferedReader(new FileReader(
                FILENAME));
```

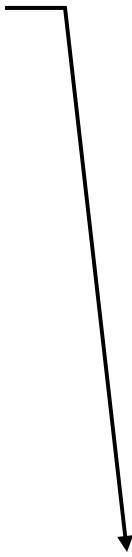


```

//The line holds the line read by BufferedReader
String line;
// The string tokenizer helps us separate the data items.
StringTokenizer sepToken;
//both ready() and read() might throw IOException
while( bufReader.ready() ) {
    String part;
    // use the buffered reader to read the string till \n
    line = bufReader.readLine();
    // construct with "," as the element delimiter
    sepToken = new StringTokenizer( line, ",");
    while( sepToken.hasMoreElements () ){
        part = sepToken.nextToken();
        //Trim off the leading and trailing
        //whitespace characters.
        part = part.trim();
        //add the part to the partsList array
        partsList.add(part);
    }
}
//The Collections class works on vector objects.
//The static sort method sorts the elements found in the
//vector object.
Collections.sort(partsList);
String output = "";

```

Collections
 method
 sort()
 allows us
 to sort the
 contents of
 the Vector.



```

        for(int i = 0; i < partsList.size(); ++i)
        {
            //We get each element from the list and tack on a \n
            output += partsList.get(i) + "\n";
        }
// close the Buffered Reader
bufReader.close();

JOptionPane.showMessageDialog(null,output, FILENAME,1);

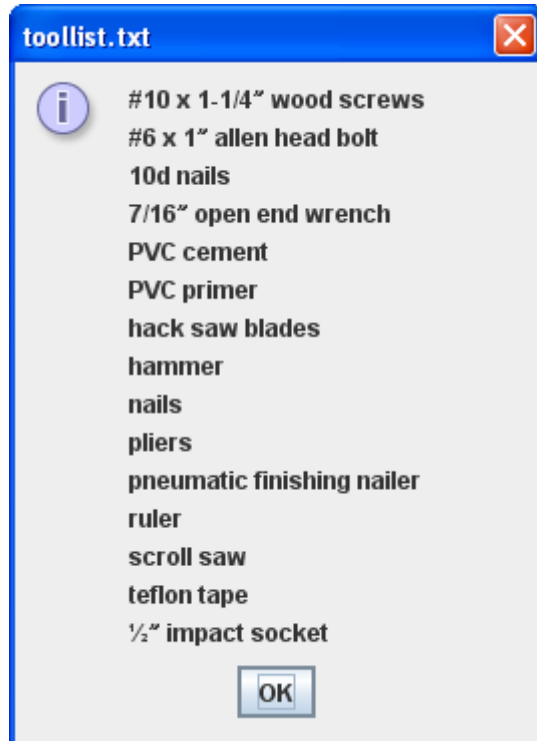
}
catch(FileNotFoundException fnfe) {
    JOptionPane.showMessageDialog(null, "Can't find the file!",
        FILENAME, 2);
    exitCode = 1; //had a problem
}
catch(IOException ioe) {
    JOptionPane.showMessageDialog(null, "Trouble!", FILENAME, 2);
    exitCode = 1;
}

System.exit( exitCode);
}
}

```



Output from `readTools.java`



Notice that the invocation of the Collections method `sort()` has produced a sorted list of tools.



File Output with the BufferedWriter Class

- The `BufferedWriter` class works in a similar manner to the `BufferedReader` class.
- A `FileWriter` object is wrapped in a `BufferedWriter` class which makes it possible to write Strings to an output file.
- The `BufferedWriter` constructors require a `FileWriter` object. The `FileWriter` object is created and then used in the constructor for the `BufferedWriter`.



File Output with the BufferedWriter Class (cont.)

- As we've seen before, this can be broken up into two separate steps:

```
//The BufferedWriter wraps the FileWriter object
//This allows us to write a data file one line at a time
FileWriter writer = new FileWriter();
BufferedWriter bufWriter = new BufferedWriter( writer);
```

- As before, the more common and preferred technique is to combine this into one line:

```
BufferedWriter bufWriter = new BufferedWriter (new FileWriter());
```



BufferedWriter Methods

void close(): Closes the file and releases any resources associated with the output stream.

void flush(): Flushes any characters out of the output stream.

void newline(): Writes a line separator into the output stream.

void write(char[] buf, int offset, int length): Writes a portion of a character array, beginning at the offset and writing *length* number of characters.

void write(char c): Writes a single character.

void write(String s, int offset, int length): Writes a portion of the String, beginning at the offset character. Writes *length* number of characters.



File Output with the BufferedWriter Class (cont.)

- In the next example program, we'll read the names of bicycles from the file `bikes.txt` and write all of the bikes in which are named Colnago into a output file named `colnagos.txt`.
- The input file (shown on page 60) contains one bike name per line.
- In the program we create `BufferedReader` and `BufferedWriter` objects and then read each line, searching it with the `String` class's `indexOf()` method. The `indexOf()` method returns the location of the substring in the `String` and a `-1` if it cannot find the substring. If we locate a Colnago bike, we'll write its entire name to the output file using the `BufferedWriter` class's `write()` method that accepts strings as input.



File Output with the BufferedWriter Class (cont.)

- The `write()` method we're using writes a portion of the String and requires the beginning position and the number of characters to be written.
 - Since we want to write the entire string, we'll use an offset of 0 and `bikeLine.length()` for the number of characters to be written.



FindColnagos.java

```
//File: FindColnagos.java    uses data file bikes.txt
//We search through the list of bikes that we read
//from the bikes.txt file looking for Colnagos.
//Write all the Colnagos to colnagos.txt file.
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.FileNotFoundException;
import javax.swing.JOptionPane;

public class FindColnagos {
    public static void main( String[] args) {
        final String FILENAME = "bikes.txt";
        final String FILEOUT = "colnagos.txt";
        int exitCode = 0;
        try {
            //FileNotFoundException thrown if we can't find the file.
            BufferedReader bufReader = new BufferedReader( new FileReader(
                FILENAME) );
            //Create a BufferedWriter object to write our Strings
            BufferedWriter bufWriter = new BufferedWriter( new
                FileWriter( FILEOUT) );
```

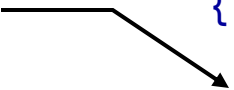


```

//The line holds the line read by BufferedReader
    String bikeLine;
    int colnagoPosition;
//both ready and read might throw IOException
    while( bufReader.ready() ) {
        // use the buffered reader to read the string till \n
        bikeLine = bufReader.readLine();
        colnagoPosition = -1;
        //The indexOf() needs exact String, returns the position if
        //it finds it
        colnagoPosition = bikeLine.indexOf("Colnago");
        if(colnagoPosition >= 0) //we have a Colnago bike
        {
            //inputs to write(String,startWritingAt,howManyChars)
            bufWriter.write(bikeLine, 0, bikeLine.length() );
            //write a newline into the file
            bufWriter.newLine();
        }
    } //end while
// close the Buffered Reader and Writer
    bufReader.close();
    bufWriter.close();
}

```

uses the last
form shown on
page 54

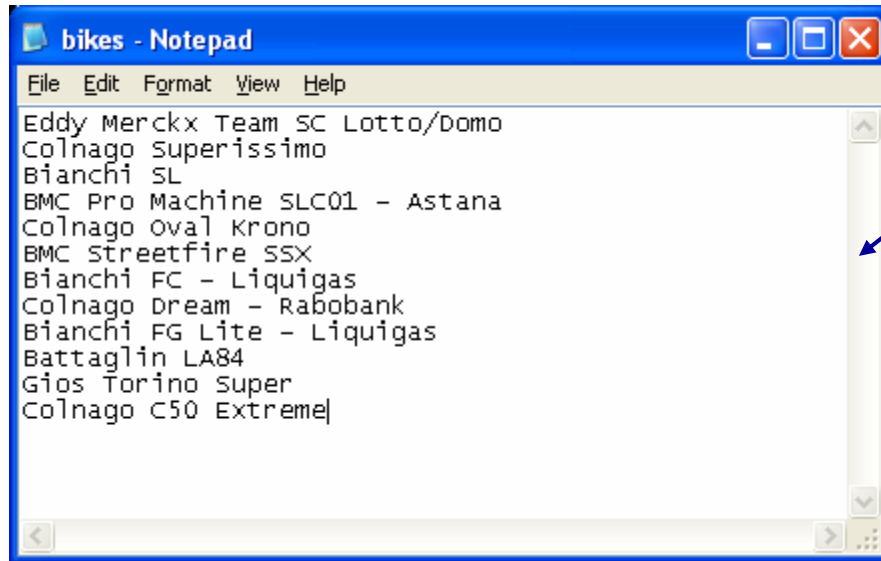



```
        catch(FileNotFoundException fnfe) {
            JOptionPane.showMessageDialog(null, "Can't find the file!",
                FILENAME, 2);
            exitCode = 1; //had a problem
        }
        catch(IOException ioe) {
            JOptionPane.showMessageDialog(null, "Trouble!", FILENAME, 2);
            exitCode = 1;
        }

        System.exit( exitCode);
    }
}
```

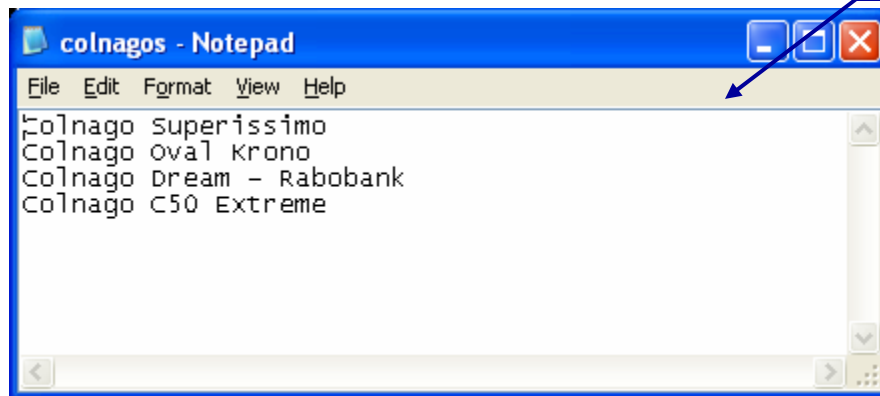


Output from FindColnagos.java



```
File Edit Format View Help
Eddy Merckx Team SC Lotto/Domo
Colnago Superissimo
Bianchi SL
BMC Pro Machine SLC01 - Astana
Colnago Oval Krono
BMC Streetfire SSX
Bianchi FC - Liquigas
Colnago Dream - Rabobank
Bianchi FG Lite - Liquigas
Battaglin LA84
Gios Torino Super
Colnago C50 Extreme
```

Input file: "bikes.txt"



```
File Edit Format View Help
Colnago Superissimo
Colnago Oval Krono
Colnago Dream - Rabobank
Colnago C50 Extreme
```

Generated output file:
colnagos.txt



Final Example Program

- Our final example is a program that includes the use of buffered readers and writers, string tokenizers, and exception handling.
- The purpose of this program is to read a file containing trip expenses and sum the various items and write the results to an output file.
 - Input file: **tripexpenses.txt**
 - Generated output file: **totaltripcost.txt**



Final Example Program (cont.)

- The input file `tripexpenses.txt` contains the following information.

```
# denotes a comment and is ignored
# This list the items and costs for a trip
# Airline tickets
$1975.00
# Rental Car
$379.99
# Gas for the rental car
$68.00
# Hotel for 3 nights
$190.18 $190.18 $179.74
# Meals for 3 days
$89.68 $189.90 $78.50
# Parking garage fees
$9.50 $9.50 $2.75
# Toll fees
$3.75 $3.75
# uncomment the next line to cause an error
# Movie tickets
```



Final Example Program (cont.)

- The program reads a line at a time and any line beginning with a “#” is not processed. The program will assume that if the line does not begin with a “#”, then it is a line that contains expense items written with the “\$” as the delimiter, and the object is searched for tokens.
 - Remember that the “\$” is not part of the extracted part of the line, and the `nextToken()` method pulls the data from between the delimiters.
- Once we’ve extracted a String containing a numeric value, we’ll use the `parseFloat()` method to convert the value to a float.
- The numeric values will be summed and the total value is written to the output file.



Final Example Program (cont.)

- I've included a bunch of JOptionPane message boxes after each method call to trace how the program is executing.
- I would encourage you to play around with this program. For example, see how many different ways you can get it to throw an exception.



TripExpenses.java

```
//File:TripExpenses.java,uses data file tripexpenses.txt creates totalcost.txt
// Reads in the expense amounts from the file, strip off the
// $ and parse the float value from the String. If the line contains
// a # in the first character, we assume it is a comment and ignore it.
// This program has one try and many catch statements. The finally block
// reports the results of the program.
// We keep count of the successful methods and write a final report
// in the finally block.
import java.io.*;
import java.util.StringTokenizer;
import javax.swing.JOptionPane;

public class TripExpenses {
    BufferedReader reader;
    BufferedWriter writer;
    static int exitCode;
    static final String FILENAME = "tripexpenses.txt";
    static final String FILEOUT = "triptotalcost.txt";

    public static void main(String[] args) {
        TripExpenses app = new TripExpenses();
        int successfulMethods = 0; //if we get 5, all is well
```



```
try {
    // open the file for reading
    app.openFileToRead();
    successfulMethods++;
    JOptionPane.showMessageDialog(null,
        "Input File opened successfully.\nAttempting to read data.",
        FILENAME,1);
    //Read the file parsing out the $ and tally the bill
    //If there is a problem with the read, we catch it here.
    float billAmount = app.readFile();
    successfulMethods++;
    JOptionPane.showMessageDialog(null,
        "Read the data successfully", FILENAME, 1);
    // open file for writing
    app.openFileToWrite();
    successfulMethods++;
    JOptionPane.showMessageDialog(null,
        "Opened the output file successfully", FILEOUT, 1);
    //Write the total expense amount to the output file.
    app.writeTripTotal(billAmount);
    successfulMethods++;
    JOptionPane.showMessageDialog(null,
        "Have written the output file successfully", FILEOUT, 1 );
    // close the files
    app.closeFiles();
}
```



```

        successfulMethods++;
        JOptionPane.showMessageDialog(null,
            "Close the files successfully", FILENAME+" " +FILEOUT, 1 );
    }
    catch(IOException ioe){
        JOptionPane.showMessageDialog(null,
            "File errors.\nExiting with error code 1.");
        ioe.printStackTrace();
        exitCode = 1;
    }
    catch(NumberFormatException nfe){
        JOptionPane.showMessageDialog(null,
            "The file is not in the proper format.\nExiting with error
            code 2.");
        nfe.printStackTrace();
        exitCode = 2;
    }
    // report how many methods were called
    finally {
        if(successfulMethods == 5)
            JOptionPane.showMessageDialog(null,
                "Completed all method calls successfully.");
        else
            JOptionPane.showMessageDialog(null,
                "Completed " + successfulMethods + "
                successful method call(s).");
    }

```



```

        System.exit(exitCode);
    }

    private void writeTripTotal(float total) throws IOException {
        //Use a DecimalFormat object to format our output numbers.
        //We use the package.Class name here instead of importing it.
        java.text.DecimalFormat currency = new
            java.text.DecimalFormat("0.00");
        String howMuch = currency.format(total);
        writer.write("Your total trip expenses amounted to $");
        writer.write(howMuch, 0, howMuch.length());
        writer.write(".");
    }

    private void openFileToRead() throws IOException {
        reader = new BufferedReader(new FileReader(FILENAME));
    }

    private void openFileToWrite() throws IOException {
        writer = new BufferedWriter(new FileWriter(FILEOUT));
    }

    private float readFile() throws NumberFormatException, IOException {
        // Holds the entire line read by BufferedReaders
        String line;
        float amount = 0.0f;
    }

```



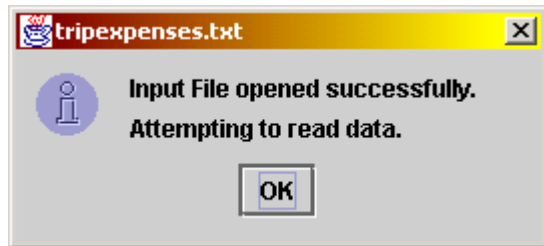
```

// The string tokenizer class allows an application
// to break a string into tokens.
StringTokenizer token;
while(reader.ready()) {
    // use the buffered reader to read the string till \n
    line = reader.readLine();
    //Line example:  # Toll Fees
    //Line example:  $3.75  $3.75
    //We want to pull out the lines with the costs.
    if( line.charAt(0) != '#'){
        //The $ is the delimiter and separates our tokens.
        //Our token will have XX.XX  form.
        //The parseFloat ignores the trailing spaces.
        token = new StringTokenizer(line, "$");
        // separate the elements
        while(token.hasMoreElements() ){
            amount += Float.parseFloat(token.nextToken());
        }
    }
}
return amount;
}
private void closeFiles() throws IOException {
    reader.close();
    writer.close();
}
}

```



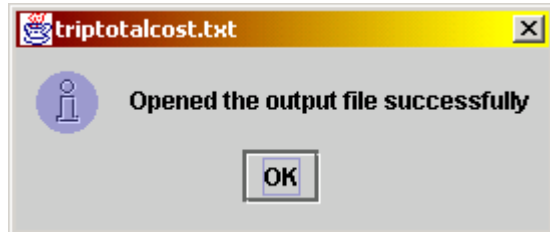
Output from TripExpenses.java



First output from
TripExpenses.java



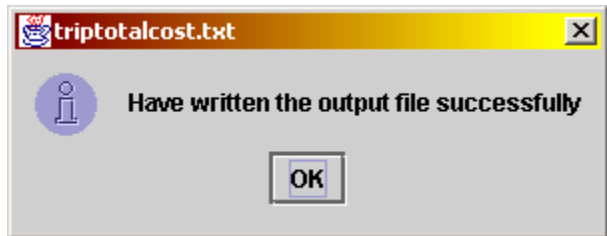
Second output from
TripExpenses.java. Notice
how the header line is displaying
the name of the input file in these
first two outputs.



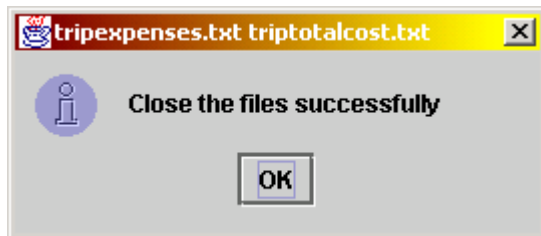
Third output from
TripExpenses.java. Notice
that the header line has changed
to reflect the output file name.



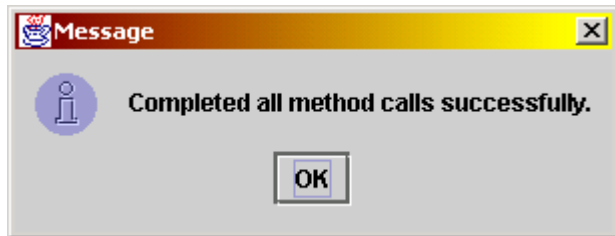
Output from TripExpenses.java (cont.)



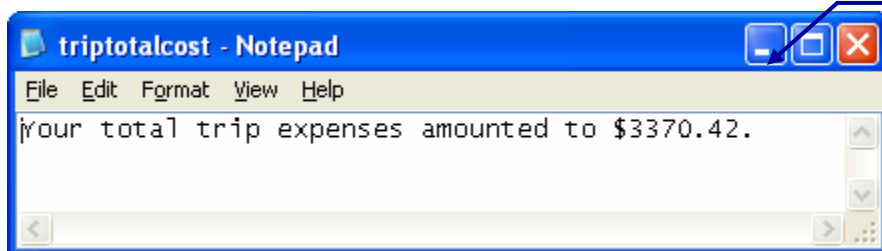
Fourth output from
TripExpenses.java



Fifth output from
TripExpenses.java.



Sixth output from
TripExpenses.java.



The file "totaltripcost.txt"
produced by the program
TripExpenses.java as displayed in
Notepad.

