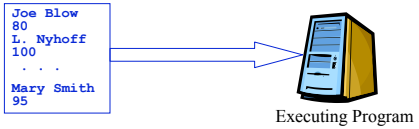
Files

Chap. 10

Streams, Readers, Writers

Problem

In our array example, we entered the students' names and scores from the keyboard. In many situations this is not practical because there is too much data. What we would like is to be able to read this data directly from a file:



Java's I/O System

- All input and output in Java is accomplished by classes called streams.
- Input streams provide ways to move bytes of data from an input device to a program.
- Output streams move bytes of data from the program to an output device.

Predefined Streams

`System` class provides three public class variables that are streams:

- **`System.in`**
  - **`InputStream`** object, usually associated with the keyboard
- **`System.out`**
  - a buffered **`PrintStream`** object, usually associated with the screen or an active window
- **`System.err`**
  - an unbuffered **`PrintStream`** object usually associated with the screen or console window

Wrapper Classes

The **`PrintStream`** class provides convenient **`print()`** and **`println()`** methods for outputting primitive type values.

Basically, all the **`Screen`** class in **`ann.easyio`** does is send these messages to **`System.out`**; e.g.,

```
System.out.println("The square root of " +
                   value + " = " +
                   Math.sqrt(value) );
```



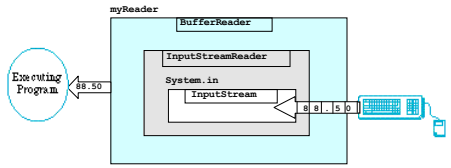
```
// From Screen.java in easy.io

/** println(double) displays a double followed
  * by a newline.
  *  Precondition:  System.out is open;
  *                   value is a double.
  *  Postcondition: value and a newline have
  *                 been appended to System.out.
  *  Return:        the receiver
  */

 public Screen println(double value)
 {
   System.out.println(value);
   return this;
 }
```


However, the **`InputStream`** class provides only methods for reading bytes.

– To read at a higher level we must "wrap" **`System.in`** with another class (a Reader class) that provides some higher-level methods (e.g., the **`BufferedReader`** class has **`read()`** and **`readLine()`** methods for reading characters and strings, respectively).

Example: **`BufferedReader`** class

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

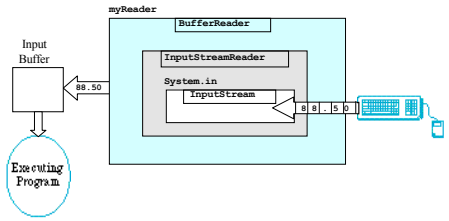


Now we can send **`myReader`** either

– the **`.read()`** message for a single char value
or

– the **`.readLine()`** message for an entire line of text

A **`BufferedReader`** is so named because it *buffers* the input, which improves program performance.



But . . .

these are the only input methods provided in class **`BufferedReader`**!

So if we need more powerful input methods — e.g., **`readInt()`**, readDouble() —

we must build them ourselves using **`read()`** and **`readLine()`**.

This is what the **`Keyboard`** class in **`ann.easyio`** does.


```
/** readDouble tries to read the next word as a double value.
  *  Precondition:  System.in is open and contains a double
  *                 value.
  *  Postcondition: The read position has advanced beyond the
  *                 next word.
  *  Return:        the double equivalent of the next word.
  *  NOTE: In earlier versions of Java that don't support
  *       parseDouble(), replace the return statement by:
  *           return Double.valueOf(myString).doubleValue();
  */
 public double readDouble()
 {
   myString = readWord();
   return Double.parseDouble(myString);
 }

   private static BufferedReader
           myReader = new BufferedReader(
                           new InputStreamReader(
                                 System.in));
```


Readers and Writers

See the Java API

- Java's current I/O system provides:
  - **`Reader`** and **`Writer`** classes: provide support for **`char`** (16-bit Unicode) I/O.
  - **`InputStream`** and **`OutputStream`** classes: provide support for **`byte`** I/O.
- **General rule of thumb:** Use a reader or writer class whenever possible. Revert to stream classes only when necessary.

Exceptions

And one more "complication" . . .

Many things can go wrong when doing I/O:

- input file doesn't exist
- invalid input data
- output file is in use or doesn't exist
- . . .

When such an error occurs, the method in which this abnormal event happened can throw an exception.

Java can catch the exception if it happens in a `try` block:

```
try {
  // call a method that may
  // throw an exception
}
```

This is followed by one or more `catch` blocks that determine the kind of exception and specify how to handle it:

```
catch (ExceptionType variable) {
  // Action to take when
  // an exception of this
  // type is thrown
}
```


General form:

```
try {
  // Call to exception-throwing method
  . . .
}
catch (ExceptionType1 variable_name1) {
  // Code to handle ExceptionType1 exceptions
}
catch (ExceptionType2 variable_name2) {
   // Code to handle ExceptionType2 exceptions
}
// ... may be more catch blocks
finally {
   // Optional finally block of
   // code to execute at the end
}
```


If the method called in the **`try`** block:

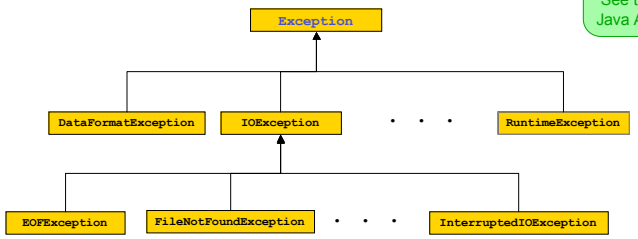
- doesn't throw an exception, control returns to the **`try`** block and continues on to the end of it, bypasses all the **`catch`** blocks, and continues with the **`finally`** block, if there is one;
- throws an exception of type ***`ExceptionType`***$_i$

  ```
  throw new ExceptionTypei();
  ```

  control is transferred to the **`catch`** block for that type, executes the code in it, and continues on to the **`finally`** block, if there is one (unless the **`catch`** block terminates execution).

The Exception Hierarchy

See the Java API



Note: A **`catch`** block for an exception of a certain type can be used for exceptions of any type *derived from* (descendant of) that type. In particular, a **`catch`** block for type **`Exception`** (the most general type at the top of this hierarchy) can catch an exception of any other type.

17

Most of Java's I/O and file-handling methods throw exceptions. In particular, **`BufferedReader`**'s **`read()`** and **`readLine()`** methods throw an **`IOException`** if an I/O error occurs.

See the Java API

This means that to use these methods, we must use the try-catch mechanism.

Example: Redo the student-grades example from the arrays section, but without using the **`ann.easyio`** package.

Note: Readers, writers, and exceptions must be imported from the **`java.io`** package.

18

```
import java.io.*;    // BufferedReader, Exception, . . .
import ann.util.*;   // Controller.fatal()

class Assignment {

  public Assignment()  {
    studentNames = null;
    studentScores = null;
    size = 0;
  }

  public double average()  {
    int sum = 0;
    for (int i = 0; i < size; i++)
      sum += studentScores[i];
    return (double)sum / size;
  }

  public void printStats() {
    double theAverage = average();
    System.out.println("\nThe average is: " + average());
    System.out.println("The deviations are:");
    for (int i = 0; i < size; i++)
      System.out.println(studentNames[i] + " "
                  + studentScores[i] + "   [" +
                  + (studentScores[i] - theAverage" + "]" );
  }
```

19

```
public void read() {
   System.out.print("Enter the size of the class: ");
   BufferedReader aReader =
       new BufferedReader(
                new InputStreamReader( System.in ));
   String numberString;

   try {
     numberString = aReader.readLine();
     size = Integer.parseInt(numberString);
     if (size <= 0)
       Controller.fatal( "Assignment.read()",
                         "Negative array size: " + size );
     else {
       studentNames = new String [size];
       studentScores = new double [size];
     }
   }
   catch (IOException exc) {
     Controller.fatal( "Assignment.read()", e.toString() );
   }
```

20

```
    String name;
    System.out.println("Enter the names and scores of "+
                       "the students in the class: ");
    for (int i = 0; i < size; i++) {
      System.out.print((i + 1) + ": ");
      try {
        studentNames[i] = aReader.readLine();
        numberString = aReader.readLine();
        studentScores[i] = Double.parseDouble(numberString);
      }
      catch (IOException ioe) {
        Controller.fatal( "Assignment.read()",
                          "Input error reading student info" );
      }
    }
  }

  private int size;
  private String [] studentNames;
  private double [] studentScores;

} // end of class Assignment
```



```
class Teacher1 {
  public static void main(String [] args)  {
    Assignment theAssignment = new Assignment();
    theAssignment.read();
    theAssignment.printStats();
  }
```

Sample run:

```
Enter the size of the class: 3
Enter the names and scores of the students in the class:
1: Joe Blow
80
2: L. Nyhoff
100
3: Mary Doe
90

The average is: 90.0
The deviations are:
Joe Blow 80.0  [-10.0]
L. Nyhoff 100.0  [10.0]
Mary Doe 90.0  [0.0]
```


Reading from a File

- We use a **`FileReader`** class to build a stream from a file to our program by sending the name of the file to its constructor:

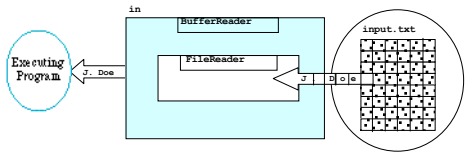
  `FileReader(`*`input_filename`*`)`

  However, **`FileReader`** has no methods to read numbers or even **`String`** values from the file its **`read()`** method only reads a single **`char`** value.

- But ...
  the **`BufferedReader`** class has:
  - a **`readLine()`** method that can read **`String`** values,
    and
  - a constructor we can use to wrap a **`BufferedReader`** around any Reader, in particular, around a **`FileReader`**.

So we build an input stream from the file to our program with:

```
BufferedReader in =
  new BufferedReader(
    new FileReader(input_filename) );
```



Now, the **`BufferedReader in`** can be send a **`readLine()`** message:

```
valueString = in.readLine();
```


- How does one know when all the data in a file has been read?
  - When no data is read, the **`readLine()`** method returns the value null; **`read()`** returns -1

```
valueString = inFile.readLine():
while (valueString != null)
  {   … // process valueString
    valueString = inFile.readLine();  }
```

- Then close the reader:

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

- It's also possible to check for other things such as an empty line with no text:

```
if (valueString.equals("")) continue;
```

Go to Slide 27

Summary of how to Read from a File

1. Build a **`BufferedReader`** ***`aReader`*** by wrapping one around a **`FileReader`**:

```
BufferedReader inFile =
    new BufferedReader(
        new FileReader( input_filename ));
```

   - This can throw a **`FileNotFound`** exception, so do this in a **`try`** block; this is a special kind of **`IOException`** so we can just catch an **`IOException`** (or an **`Exception`**).
   - The name of the file can be:
     - "Hard-wired": Use "***`name_of_file`***"
     - Input into a **`String`** variable
     - Entered into **`arg[0]`** from the command line.

2. Use an input loop to read from ***`aReader`***; e.g.,

```
valueString = aReader.readLine():
while (valueString != null) {
  // process valueString
  valueString = aReader.readLine();
}
```

   - This can throw an **`IOException`**, so do this in a **`try`** block.
   - Convert ***`valueString`*** to numeric values if necessary.

3. Close ***`aReader`***:

```
aReader.close();
```

Writing to a File

1. Build a PrintWriter object connected to the output file. For this, we need three classes:
   - **`FileWriter`** to construct an output stream to the file
   - Wrap this in a **`BufferedWriter`** to improve output efficiency
   - Wrap this in a **`PrintWriter`**, which provides **`print()`** and **`println()`** methods.


```
PrintWriter aWriter =
  new PrintWriter(
    new BufferedWriter(
      new FileWriter( outFilename )));
```

An IOException can occur, so this must be done in a **`try`** block.

2. Use **`aWriter.print()`** and **`aWriter.println()`** to write output to the file.
3. Close the file:

```
aWriter.close();
```


Example: Redo the student-grades example from the arrays section, but with file I/O.

```
import java.io.*;       // BufferedReader, FileReader, . . .
import ann.util.*;      // Controller

class Assignment {

  public Assignment()  {
    studentNames = null;
    studentScores = null;
    size = 0;
  }

  public double average()  {
    int sum = 0;
    for (int i = 0; i < size; i++)
      sum += studentScores[i];
    return (double)sum / size;
  }
```



```
public void printStats(String outFilename)  {
  try {
    PrintWriter aWriter =
              new PrintWriter(
                  new BufferedWriter(
                      new FileWriter( outFilename )));
    double theAverage = average();
    aWriter.println("\nThe average is: " + average());
    aWriter.println("The deviations are:");
    for (int i = 0; i < size; i++)
      aWriter.println(studentNames[i] + " "
                  + studentScores[i] + "   [" +
                  + (studentScores[i] - theAverage" + "]" );
    aWriter.close();
  }
  catch (IOException ioe) {
      Controller.fatal("Assignment.printStats()",
                       ioe.toString());
  }
}
```

```
public void read(String inFilename) {
  String numberString;
  try {
    BufferedReader aReader =
        new BufferedReader(
            new FileReader( inFilename) );

    numberString = aReader.readLine();
    size = Integer.parseInt(numberString);

    if (size <= 0) {
      aReader.close();
      Controller.fatal("Assignment.read()",
                       "Illegal array size: " + size);
      //-- or we could throw an exception
    }
    else {
      studentNames = new String [size];
      studentScores = new double [size];
    }
```

Can throw NumberFormatException


```
    for (int i = 0; i < size; i++) {
      studentNames[i] = aReader.readLine();
      numberString = aReader.readLine();
      if (studentNames[i] == null
            || numberString == null) {
        aReader.close();
        Controller.fatal("Assignment.read()",
                         "Out of data for student " + i);
        //-- or we could throw an exception
      }
      studentScores[i] = Double.parseDouble(numberString);
    }
    aReader.close();
  }
  catch (Exception e) {
    Controller.fatal("Assignment.read()", e.toString());
  }
}

private int size;
private String [] studentNames;
private double [] studentScores;

} // end of class Assignment
```

Can throw NumberFormatException


```
class Teacher3 {
 public static void main(String [] args)  {

   if (args.length < 2)
     Controller.fatal("main(): ", "Missing file name");

   String inFilename = args[0],
          outFilename = args[1];
   Assignment theAssignment = new Assignment();
   theAssignment.read(inFilename);
   theAssignment.printStats(outFilename);
 }
}
```



```
% cat scores.txt
3
Joe Blow
80
L. Nyhoff
100
Mary Q. Doe
90

% java Teacher3 scores.txt scores.out

% cat scores.out

The average is: 90.0
The deviations are:
Joe Blow 80.0  [-10.0]
L. Nyhoff 100.0  [10.0]
Mary Q. Doe 90.0  [0.0]
```

Suppose we throw the exceptions in **`Assignment`**'s **`read()`** as described in the comments:

```
public void read(String inFilename)
  throws EOFException, IOException  {

  String numberString;
  try {
    BufferedReader aReader =
        new BufferedReader(
             new FileReader(inFilename) );

    numberString = aReader.readLine();
    size = Integer.parseInt(numberString);

    if (size <= 0) {
      throw new IOException(
                "IIlegal array size: " + size);
    }
    else {
      studentNames = new String [size];
      studentScores = new double [size];
    }
```



```
    for (int i = 0; i < size; i++) {

      studentNames[i] = aReader.readLine();
      numberString = aReader.readLine();
      if (studentNames[i] == null
             || numberString == null) {
        throw new EOFException(
                  "Out of data for student " + i);
      }
      studentScores[i] = Double.parseDouble(numberString);
    }
    aReader.close();
  }
  catch (IOException ioe) {
    Controller.fatal("Assignment.read()", ioe.toString());
  }
}
```


And change **`main()`** to:

```
public static void main(String [] args)  {

  if (args.length < 2)
    Controller.fatal("main(): ", "Missing file name");

  String inFilename = args[0],
         outFilename = args[1];
  Assignment theAssignment = new Assignment();
  try {
    theAssignment.read(inFilename);
    theAssignment.printStats(outFilename);
  }
  catch (EOFException e) {
    System.out.println(e.toString());
  }
  catch (IOException e) {
    System.out.println(e.toString());
  }
}
```

or just

```
catch (Exception e) {
 System.out.println(e.toString());
}
```



```
% cat scores1.txt
4
Joe Blow
80
L. Nyhoff
100
Mary Q. Doe
90
% java Teacher4 scores1.txt out
*** Assignment.read(): java.io.EOFException:
                                  Out of data for student 3
% cat scores2.txt
-1
Joe Blow
80
L. Nyhoff
100
Mary Q. Doe
90
% java Teacher4 scores2.txt out
*** Assignment.read(): java.io.IOException:
                                  Illegal array size: -1
```

Binary Files

Readers and Writers use text-based I/O in which each character is stored using 2 bytes; e.g, 2147483647 requires 20 bytes. Storing its 32-bit binary representation,

01111111 11111111 11111111 111111111

would require only 4 bytes.

Java's Stream classes can be used for such binary I/O. Two of these are **`DataInputStream`** and **`DataOutputStream`**. They contain methods — e.g., **`readDouble()`**, **`writeDouble()`** — for reading and writing binary data. (See Fig.10.3 for a demo.)


```
/** DataStreamDemo.java demonstrates DataOutputStreams and DataInpusStreams.
 *  Ouptut (file) : a sequence of doubles to a file via a DataOutputStream
 *  Input(file)   : the doubles from this file via a DataInputStream
 *  Output(screen): the sequence of doubles
 */

import ann.util.*;       // Controller
import java.io.*;        // DataOutputStream, DataInputStream...

class DataStreamDemo extends Object
{
 public static void main(String [] args)
 {
   DataOutputStream dataOut = null;
   DataInputStream dataIn = null;

   try
   {
     dataOut = new DataOutputStream(
                new BufferedOutputStream(
                 new FileOutputStream("numbers.dat")));

     for (double val = 1; val <= 10000; val *= 10)
       dataOut.writeDouble(val);

     dataOut.close();
```

```
    try
    {
      dataIn = new DataInputStream(
                new BufferedInputStream(
                 new FileInputStream("numbers.dat") ));
      double value;
      for (;;)
      {
        value = dataIn.readDouble(); // EOFException breaks if no more data
        System.out.println(value);
      }
    }
    catch (EOFException e)
    {
      dataIn.close();
      System.out.println("Processing complete.\n");
    }
  }
  catch (Exception e)
  {
    Controller.fatal("main()", e.toString());
  }
 }
}
```

```
holmes ~/cs185/classprogs$ java DataStreamDemo
1.0
10.0
100.0
1000.0
10000.0
Processing complete.

holmes ~/cs185/classprogs$ cat numbers.dat
?>@$@Y@@@Ãholmes ~/cs185/classprogs$
```