File Input/output using Loops

Frequently a program will require more input data than can simply be input by a user at the keyboard. Data files can be prepared ahead of time using a text editor, stored on disk, and accessed via an executing program. Also, instead of writing output to the screen, a program can direct that that output be stored in a file and saved to disk.

There are many classes provided by Java that help us read from & write to **text** files. The simplest Classes to use are classes **Scanner** and **PrintWriter**.

Reading input from a File

To read data from a file, the programmer needs to have an understanding of the **format of the data stored in the file, and its meaning.** Sometimes a file might simply contain a series of values (of unknown length) that must be processed one at a time for example: a series of integers whose sum and average must be calculated.

```
99  23  12  4 15 12 147 125 123 88 97 96 425
12 11 1 0 -52 100 310 354 56
```

A file might also contain lines of text that must be read one line at a time and processed in special ways. For example a data file might contain lines of text, and a program will need to read in each line counting the number of words in the whole file:

```
Once upon a midnight dreary, while I pondered, weak and weary,
Over many a quaint and curious volume of forgotten lore,
```

While others might contain **logical records.** Here each physical line in the file will contain **multiple** values which **together** form a **logical record**. A **logical record** is a set of values which together describe one unique entity in our problem space. For example if we have a data file containing information about employees and their hours worked used to compute their weekly pay. Such a file might look like the following:

Smith	**John**	**37.5**	**7.35**	**s**	**b**
Lewis	**Larry**	**45.0**	**5.50**	**s**	**n**
Jones	**James**	**55.0**	**8.20**	**m**	**b**

In this example, the first string represents the employee's **last name,** followed by **first name**, **number** of **hours worked**, **hourly wage**, whether **single** or **married**, and whether or not they are **benefits** eligible.

So the first line would translate into:

John Smith, worked 37.5 hours, at $7.35 per hour, he is single, and is benefits eligible.

No matter what the data file contains we need write our program to read and process the data the file contains either one value at a time, one line at a time, or one logical record at a time. To do this we use **loops**.

The role of class Scanner:

In general when we access data contained in a file we DO not know HOW many values are there. Class **Scanner** offers many methods that you can use with **loops** to **read** multiple "**lines**" of data from a data **file OR to check if any input stream contains MORE values of a certain type**. In addition to the methods that accomplish general input such as nextInt(), nextFloat() etcetera that would be used to access individual values, Class Scanner offers others that manipulate and test the contents of an input stream to see if there are more values available to read, these include:

- **hasNext():** This method returns true if the input stream has more data of any type that has not yet been read.
- **hasNextInt():** returns true if the next value in the input stream can be interpreted as an Integer.
- **hasNexttFloat():** returns true if the next value in the input stream can be interpreted as an floating point number.
- **hasNextBoolean():** returns true if the next value in the input stream can be interpreted as a Boolean value.
- **hasNextLine():** returns true if there is another line of input in the input stream (file) terminating with a carriage return .

Using one of these methods, as we can create a loop to read data from an input stream, **until all of that data is exhausted.**

For example, if we have defined a **Scanner** object, called **inputFile** the logic of our program would look like:

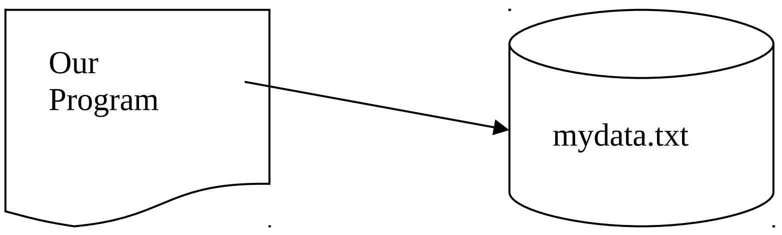
```
  while (inputFile.hasNext()) {

          //READ THE DATA FROM THE INPUT STREAM

          // PROCESS THE DATA
}
```

We could also use other has methods as appropriate to the data we are reading, although has next is more general and works with files containing logical records.

Attaching a Scanner to a data file

We can not attach a scanner directly to a data file, instead to allow Scanner to access a physical file, we must use class **FileReader or class File to attach to the file, and then create a Scanner based on our instance of class FileReader**. Instances of this class are used to create a connection between our program and a physical file. For example if we are trying to read a file called mydata.txt we can access it via.

Scanner infile = new **Scanner**(new **FileReader**("c:\\mydata.txt"));



Any methods from Scanner that are invoked for "**infile**" are now applied to the data file, rather than the console. In the previous example the instance of FileReader is created when we create the instance of scanner, there are other ways:

FileReader *name* = new FileReader("*path and file name*");

We can define instance of FileReader separately, the filename given may or may not include the full path, if only a file name is given ie: "***mydata.txt***" then the file is assumed to be in the current working directory which is the one containing your source code. You will notice something unusual in the first example which included a path name. The character \ is a control character in Java, as in *\n* which is the new line character. If we need to give a fully qualified path name, then we must use two \\. So for example: ***"c:\\cs110\\inputdata\\mydata.txt".***

Once the instance of FileReader is declared, we can then use it in the declaration of our scanner:

FileReader *inputFile* = new FileReader("*mydata*.*txt*");
Scanner *infile* = new Scanner(*inputFile*);

Please Note: when declaring an instance of **FileReader**, the file name can also be specified using a string variable, instead of a quoted literal. Which means it is possible to **prompt** the user for the name of the input file, read it into a string variable, and then create our connection to the file.

String filename;
Scanner console = new Scanner(System.in);

System.out.println("Please enter the name of the file to be read. ");
filename = console.nextLine();

FileReader inputFile = new FileReader(filename);
Scanner input = new Scanner(inputFile);

This leads to another idea, a program can have MULTIPLE Scanners one for input from the keyboard and others that access the contents of data files.

Error Handling Essentials

There is a possible error condition for which we must account in case the file we are accessing does not exist. This is a "**file not found**" exception, and this causes Java to **raise** or "**throw**" an **IOException**. What does that mean anyway????

There are two types of exceptions thrown by Java methods: **checked and unchecked**. **An exception is an error that occurs during the execution of a program**.. When you **WRITE CODE THAT** calls a method that throws a "**checked**" exception, the compiler checks to see that you don't ignore it!!!!!!! YOUR program must contain code that deals with the exception.   "**Checked**" exceptions are caused by external circumstances that the programmer cannot control or prevent but **must plan for**. Therefore your program must provide a means of recognizing that these can occur and respond to them. Class **FileReader** throws **checked** exceptions. An **unchecked** exception typically arises from a mistake made by the programmer such as trying to read **past the end of a string, or dividing by zero.** The compiler does not check for these types of errors and cannot.

There are two ways we can deal with **checked** exceptions:

- We can write exception handlers which are special sections of code that are invoked when the error occurs,
- we can choose to simply pass the error "back", calling routine, which in this case to the operating system or java virtual machine, and terminate. We do this through using the "**throws**" specifier in our method heading for method **main**.

```
import java.io.*;

public class readingFiles1 {
   public static void main(String[] args) throws IOException {

          File infile = new File("c:\\cs110\\java\\indat.txt");
          FileReader in = new FileReader(infile);

   }
}
```

If the file doesn't exit this displays, then the system prints a stack trace:

```
Exception in thread "main" java.io.FileNotFoundException:
c:\cs110\java\indat.txt (The system cannot find the file specified)
at java.io.FileInputStream.open(Native Method)
at java.io.FileInputStream.<init>(Unknown Source)
at java.io.FileReader.<init>(Unknown Source)
at readingFiles1.main(readingFiles1.java:7)
```

The "throws" specifier just indicates that you **KNOW** that this exception might occur, and to terminate your program when this happens, passing the exception back to the **CALLING** program. The caller then needs to make the same decision – either handle the exception or pass it back to it's calling program.

A better option is to include a **TRY/CATCH** block within your code to intercept the error, and attempt to correct the problem OR provide more descriptive error messages. To use a try/catch block:

- Place the code which MIGHT GENERATE the exception inside the **try** block.
- Write a **catch block** for the type of exception that might occur, which includes code that deals with the error.

```
import java.io.*;

public class readingFiles2 {
    public static void main(String[] args){

    try{
            File infile = new File("c:\\cs110\\java\\indat.txt");
            FileReader in = new FileReader(infile);
       }// end try

 catch( IOException e){
    System.out.println("You have entered an incorrect file name,"
      + " please try again with the correct name");
 }// end catch


}// end main
}// end class
```

Forcing the user to input a correct name:

It is possible to incorporate a loop with a try/catch block to force the user to input a correct file name, by simply using a Boolean variable and placing the try/catch block inside a while loop.

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

public class CorrectFile {
   public static void main(String[] args){
// this boolean variable is false until a good file name is input
            boolean goodFileName = false;
            String fileName;
            Scanner input = new Scanner(System.in);

            System.out.println("Please enter a file name.");
            fileName = input.nextLine();


            while (!goodFileName){
               try{
                     FileReader in = new FileReader(fileName);
                     Scanner dataFile = new Scanner(in);

                     // if we reach this statement then the file exists
                     goodFileName = true;
              }// end try

             catch( IOException e){
                     System.out.println("You have entered an incorrect file name,"
                                 + " please try again with the correct name");
                     System.out.println("please enter another file name");
                     fileName = input.nextLine();
             }//end catch

             }// end while

            System.out.println("You have successfully connected a scanner to a data file.");
   }// end main
}// end class
```

<u>Example of reading an input file;</u>

The following are a few examples that read a variety of text files:

1) Example ONE reads a file containing an unknown number of integers and calculates the average of the numbers contained in the file and prints the files contents, ten numbers per line

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

public class ReadNumbers{
public static void main(String[] args){
  Scanner input = new Scanner(System.in);
  String fileName;
  boolean goodName = false;
  int currentNumber, sum = 0, numberCount=0;
  Scanner numberFile = null;
  FileReader infile;

  System.out.println("Please input the name of the data file");
  fileName = input.nextLine();

  while (!goodName){
   try{
        infile = new FileReader(fileName);
        numberFile = new Scanner(infile);

        goodName = true;
 }//end try
 catch(IOException e){
        System.out.println("invalid file name, please enter another name");
        fileName = input.nextLine();
 }// end catch
 }//end while


 // we have a good file open READ the contents
 System.out.println("The numbers contained in the file are: ");

 while (numberFile.hasNextInt()){
        currentNumber = numberFile.nextInt();
        System.out.print( currentNumber + " ");
        sum += currentNumber;
        numberCount++;

        if (numberCount%10 == 0) System.out.println();//newline
 }// end second While
 System.out.println("\nThe average of the numbers in file is " + ((double) sum)/
numberCount);
}// end main
}// end class
```

2) Example two Read a file of text, one line at a time, counting the number of blank spaces in each line

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

public class ReadLlines{
public static void main(String[] args){
  Scanner input = new Scanner(System.in);
  String fileName;
  boolean goodName = false;
  int blankCount=0;
  String currentLine;
  char c;
  Scanner lineFile = null;
  FileReader infile;

  System.out.println("Please input the name of the data file");
  fileName = input.nextLine();

  while (!goodName){
try{
     infile = new FileReader(fileName);
     lineFile = new Scanner(infile);

     goodName = true;
}//end try
  catch(IOException e){
     System.out.println("invalid file name, please enter another name");
     fileName = input.nextLine();
}// end catch
 }//end while

 // we have a good file open READ the contents
 System.out.println("The Lines contained in the file are: ");
 while (lineFile.hasNextLine()){

   currentLine = lineFile.nextLine();
     System.out.println(“The current line is: “ + current);

     for(int j=0;  j< currentLine.length();  j++){

          c = currentLine.charAt(j);
          if (c == ' ') blankCount++;
    }// end for

  System.out.println(currentLine);
  }// end second While

 System.out.println("\nThe file contained " + blankCount + " blank 
spaces.");
}// end main
}// end class
```

3) Write a program to read employee information from a data file, and calculate the employee's weekly pay and withholdings using the following formula:

grossPay = hours worked * hourly rate

Tax = grossPay * taxRate

If employee is married, taxRate = 17%, if single 13%

If the employee is benefits eligible,

benefitsCost = grossPay * 3%.

netPay = grossPay –(benefitsCost + tax);

The format of the data stored on EACH line of the data file is :

lastname firstname hoursWorked payRate taxStatus BenefitsEligible

Each value is separated by a blank space

When a files contains a line consisting of data of various types, we make a multiple calls to scanner, one for each piece of data we need to read.

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

public class PayChecks {

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

 final float SINGLERATE = .13f;
 final float MARRIEDRATE = .17f;
 final float BENEFITSRATE = .03f;

 String last, first;
 char benefits, taxCode;
 float hoursWorked, rate, netPay, grossPay, tax=0,
        benefitsCost, taxRate;

 FileReader inputFile = new FileReader("employee.txt");
 Scanner input = new Scanner(inputFile);

 System.out.printf("%-10s %-10s %-10s %-10s %-10s %-10s %-10s %-10s\n",
"Last", "First", "Hours", "Pay", "Tax", "Benefits", "Gross", "Net");

System.out.printf("%-10s %-10s %-10s %-10s %-10s %-10s %-10s %-
10s\n\n","Name", "Name", "Worked", "Rate", "Cost", "Cost", "Pay", "Pay");

 while (input.hasNextLine()) {
         last = input.next();
        first = input.next();
        hoursWorked = input.nextFloat();
        rate = input.nextFloat();
        taxCode = (input.next()).charAt(0);
        benefits = (input.next()).charAt(0);

        grossPay = hoursWorked * rate;

        if (taxCode == 's' || taxCode == 'S') tax = grossPay *SINGLERATE;
        else if (taxCode == 'm' || taxCode == 'M')
                tax = grossPay * MARRIEDRATE;
        else System.out.println("INVALID TAXCODE");

        if (benefits == 'b' || benefits == 'B')
                benefitsCost = grossPay * BENEFITSRATE;
        else benefitsCost = 0;

        netPay = grossPay - (benefitsCost + tax);
System.out.printf("%-10s %-10s %-10.2f %-10.2f %-10.2f %-10.2f %-10.2f %-
10.2f\n", last, first, hoursWorked, rate, tax, benefitsCost, grossPay,
netPay);__

 }// end of while
  }// end main
 }// end class
```

When run with the data file:

```
Smith  John   37.5   7.35   s  b
Lewis Larry   45.0  5.50   s  n
Jones  James   55.0 8.20   m  b
```

The output is :

```
Last      First     Hours      Pay       Tax       Benefits   Gross      Net
Name        Name        Worked      Rate       Cost       Cost       Pay        Pay

Smith      John      37.50      7.35      35.83      8.27       275.63     231.52
Lewis      Larry     45.00      5.50      32.17      0.00       247.50     215.32
Jones      James      55.00      8.20      76.67      13.53      451.00     360.80
```

Writing Data to an output file

The simplest method to write to a text file using stream I/O is by using the class PrintWriter. We declare an instance of PrintWriter by:

PrintWriter identifier = new PrintWriter(file name string or file instance);

ie: PrintWriter out = new PrintWriter("c:\\cs110\\java\\output.dat");

File outFile = new File("c:\\cs110\\java\\output.txt");
PrintWriter output = new PrintWriter(outFile);

This opens an output stream associated with the indicated file. IF the file already exists it is **emptied** before new data is written into it. IF it does not exist it is created.
PrintWriter allows us to use the **print**, **println**, and **printf** methods with which we are already familiar. Other methods we should be familiar with are:

- **close()** -- This methods closes the output stream. It must be performed on an output file before the method terminates, otherwise the output file may be corrupted.
- **flush()** -- Flushes any remaining data in the internal buffer to the output file.

```
import java.io.*;

public class testOutput2 {
  public static void main(String[] args) throws IOException {

   File outFile = new File("e:\\cs110\\testoutput.txt");

   PrintWriter out = new PrintWriter(outFile);

   char aChar = 'A';


// The following loops prints out a series of 10
// lines each line will print a single character
// 30 times

        for (int j=0;  j < 10; j++){
                // print the characters to the output file
           for(int i=0; i<30; i++)  out.print(aChar);
        // print a new line
           out.println();

           aChar++;// go to the next character
        } // end for

// flush sends all output to the physical file
        out.flush();
// close the file or the data will not be saved
        out.close();

  }// end main

}
```

Sample output

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAA
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBB
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCC
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDD
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EE
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```