JAVA FILE HANDLING

# INTRODUCTION

Now that you are becoming comfortable with designing and using classes, we are going to tackle the second major topic of the semester: the care and feeding of arrays. The array data structure is a very important part of advanced programming. In fact, you will soon be writing programs that incorporate an array as part of the class itself. And to provide the data for your array(s), you will be required to load values from a data file on an external storage device.

Before starting our discussion of arrays, then, we’ll focus on the use of Java classes for file input and file output. We will discuss the **FileInputStream** and **Scanner** classes to allow input from data files, and the **PrintWriter** and **FileWriter** classes to write data to an output file. Note: these classes are only briefly mentioned in the text, or are covered in a way that could be confusing to first-time users. Accordingly, the text reading schedule does not list those sections as required reading.

# SEQUENTIAL DATA FILE BASICS

Before getting your hands dirty with actual files, it’s quite important to understand how programmers (and database administrators) consider data that is stored on disk. In order to competently handle large amounts of data coming in from an external storage device, whether it be a hard drive, a flash drive, or even an Internet site, it’s very important to understand how data is organized. If you have taken a course in databases, you have most likely studied the data modeling idea, but may not have comprehended its importance until now.

In data files and databases the data itself is usually organized around a model known as the “data hierarchy”. The hierarchy starts out at its lowest level with the “smallest” piece of data that could possibly interest us. Since we don’t care about anything smaller than a single character, we think of that as the basis of the entire picture. Then, higher levels are built on the layer just below it. Consider this diagram:

**DATA HIERACHY**

Logically related group of characters

Logically related group of fields

Logically related group of records

Database

File

Field

Record

Character

As we begin reading characters out of a file, which consists only of ASCII characters, how is it possible to understand what the file contains? The answer is that we’ll think of the file as divided into logical parts and pieces. The smallest piece we can read will be one character. And if the piece of data we are reading is expected to be someone’s middle initial, then a single character will make sense.

But it’s most often the case that a data file will contain much more involved pieces of data. For example, suppose the file contains these two “lines”:

123456 Abernathy Amber A 120345.0

234567 Baker Bobby O 33294.90

A key idea to keep in mind is that this file, and all the files you will work with in this class, is a **sequential access** (ASCII) file. That means that it consists entirely of ASCII characters that are grouped together in a logical fashion. Furthermore, it means that we must read the file from the very first character down to the last character. So, when we first encounter anything from the file, the first data to be encountered are the characters 123456. Now, if we read only one character, the first 1, that would certainly not tell us much. It’s obvious that all six characters “belong” together, since they represent some sort of ID or whatnot.

That sort of “collection” is known as a **field**, which as depicted on the hierarchy diagram is a logically related set of characters. So, fields build on top of the character level. If you remember your database study, a field is given a unique name and a data type, and that’s the very same thing we will do in our programs. A field is read all at once and placed into a variable, which must match the type of data coming in. Therefore, it’s very important to know the type of every field we are reading. And you will: for any file you are to process, you will have a description of each and every field and its type, so you can declare a variable of the correct type for it.

To proceed to the next level in the hierarchy diagram, consider the set fields that are on one line above. Isn’t it rather obvious that all of the fields pertain to the same person? What that realization entails is that whenever we read from the file, it only makes sense to read all of the fields that “belong together”, and that set is called one **record**. So, every time we read anything from the file, we will read a complete record, a complete set of data that is available to describe one person. It makes no sense to read only a partial record. Why? Because first, we may as well read in all we know about one individual…makes no sense to stop short. And second, we must read through all of the data anyway, since the file is sequential. If we stopped in the middle of a record for some strange reason, the next time we went to the file we would have to continue reading at that point.

For that reason, any time we gather input from the file, we will refer to it as reading a complete **record**.

The next level up in the hierarchy is the entire **file** itself. This level is important from the standpoint that in order to access the data, you will need to know how to ask the operating system for access to the file. Then, in order to interface with it, you will use Java classes to connect to and get data out of the file. This certainly means knowing its exact filename and extension, if applicable, and also where it resides in the file structure.

The final level on the hierarchy represents the level that was instituted when the concept of a relational database system was invented. The idea was to have multiple separate files, containing different database tables, to speed up data access. You will not work with that level in this class, though.

# JAVA FILE INPUT

[Note: we will be using the following procedures to read and write files, rather than the methods shown in the textbook.]

Let’s get started by setting up a simple problem and attacking it using what we know so far. Say we have a file of data that we want to process: a file of sales values representing numbers we have gathered from a bunch of salespeople working for our company. To be a bit more specific: our file contains an unknown number of records containing two pieces of data: first, an employee ID number, and second the sales for that person for a single month. The first field (that’s what it is called in database terminology, right?) is a string and the second is a double. One of the challenges is that we don’t know how many values made it into the file, since the number of salespeople seems to vary each month. Each record is for one person only.

Note: you should have found the Java project “Simple File Input Project” zipped up with this tutorial. You should unzip it now and import it into Eclipse so you can refer to it during this discussion.

When you work with any data file, you need to have access to a description of what it contains. That is typically called a “record description”, a list of what fields are in each record and their types. Here is the description for our file:

**RECORD DESCRIPTION (employeeSales.dat):**

|  |  |
| --- | --- |
| salesID | string |
| sales | double |

And just for “grins”, let’s look at the complete file contents:

14728 4579.00

87223 8270.50

38092 9374.75

13482 1398.30

58138 3342.00

59874 5983.98

49318 1529.95

33799 7392.55

68431 6731.10

86914 3499.95

For starters, it would be a very good idea to figure out the basic logic we need to process the file and determine the total of all the sales values. If you give any thought to reading all of the data into a rather large set of variables, do your best to squelch that idea right now! Just because we only have 10 pairs of numbers right now does not mean we may not have 1000 later on as the company grows. What we need is a sound programming logic structure that repetitively reads records, and with each record adds the sales for that record to an accumulator variable. Sound good?!

So, what we need is a loop that processes each record one at a time. The basic idea would be to read one record, add the sales to our accumulator, then repeat those two steps until the file is all taken care of. The Java file class we will use makes this a snap, because it will look ahead in the file and stop when there is nothing more to read, that is, when we run out of records.

All right, I know, I know—you need just a bit more assurance! So, let’s look at a more formal solution. This will be a pretty simple program with no classes, just enough logic to arrive at the sum of all of the sales values. Later on, in the next tutorial, we will be putting a complete file-processing program to good use: loading values into an array.

**A quick file primer.** Our filename is *employeeSales.dat*. We know what the record description says about it: it contains one string and one double on each “line”, that is, per record. But why do there appear to be “lines” in the file? If you open the file in a simple text editor, like Notepad, you’ll see that it looks very much like the contents shown on the page above. Why is that?

First of all, it’s important to know what sort of character is splitting up the fields. In this file, as with all of your data files in this class, the field **delimiter**, that is, the character between fields, will be either a space or a tab character. Further, at the end of each record, there will be a carriage return character. A carriage return inserted after every record is primarily done for “human consumption”: when the file is opened in a word processor, every record will appear on a separate line, making for a much easier picture for us to understand the data we’re looking at.

The beauty of having simple delimiters like spaces and carriage returns is that the Java file method we will use to input the data will recognize them and will stop reading when it encounters one of those characters. The input mechanism doesn’t consider “spacy” characters to be important, so it won’t input them at all. That will make it a breeze to input each field of data into a separate variable. (Of course, that also means that we won’t be able to read any field that has a space in it, but that won’t impact any of your work in this class.)

In order to process this file, then, we’ll take advantage of a class you already have some experience with, the **Scanner** class. In previous programs we used that class to input data from the keyboard. In order to deal with files on disk, we’ll need to couple that with the Java **FileInputStream** class.

To remind you, here is how we instantiated a Scanner class object in previous programs:

Scanner inputDevice = new Scanner(System.in);

The System.in object of course was the keyboard. In order to hook up with a disk file, here is how the object will be instantiated now:

Scanner infile = new Scanner(new FileInputStream("employeeSales.dat"));

Let’s break down this rather involved statement. What happens first is that the FileInputStream object is created and its constructor is called. This action is described in the Eclipse tooltip window: “Creates a FileInputStream by opening a connection to an actual file, the file named by the path name in the file system.” The object’s constructor takes one argument: a String containing the path name to the file. To satisfy that requirement, we will usually name the file directly, as a String constant. However, it would be perfectly acceptable to use a String variable as an argument. If the path is more involved, then you will need to put the entire path as the argument. With only the filename spelled out, the file itself needs to be in the project’s root directory, that is, the outermost directory for the entire project.

You may be wondering why this class is called a “Stream”. That’s because it considers the data in the file as simply a continuous stream of characters. The stream starts at the first character in the file and continues to the final character, which is a special character that terminates all files and is called the End Of File, or EOF, character. This view of a file, then, does not ascribe any sort of organization to the file whatsoever. The file is considered to be one ASCII character after another, with no idea what should be considered as separators, delimiters, etc.

Before we go on to tie in the Scanner class, it would be useful to consider one of the vagaries of programming in the Windows environment. Any time you interface with any operating system, you must consider how it handles files, and specifically any unique aspects of file names, types etc. In the case of Windows, there is a sort of conflict with how it uses the backslash (\) symbol. The backslash represents two concepts in the Windows OS: first, it is the name of the root directory for any drive; and second it acts as a file name separator. For example, suppose you had a data file called simply “data.dat” in a subdirectory called companyData. The formal Windows reference to this file would look this way:

C:\companyData\data.dat

However, if you coded that as a String in a Java statement, you would run into problems, since the backslash symbol is the start of a sequence of characters called an “escape sequence”. This sort of sequence is common in most modern languages that have inherited syntactic structures from the original C language. The essential idea is how to code characters that cannot be typed at the keyboard. You should be familiar with the escape sequence “\n”. Coded into a string, that sequence will be turned by the compiler into a carriage return.

Now, what would happen if we coded the literal reference to the file into a Java string, like this?

System.out.println (“C:\companyData\data.dat”);

Answer: the Java compiler would flag both the “\c” and the “\d” as invalid escape sequences, since it has a different idea than Windows as to how to interpret the backslash symbol. Since the compiler triggers off the backslash, it would want to consider both of those sequences as escape sequences, and there is no valid sequence for either of those two letters.

The solution is to code the backslash itself as an escape sequence: “\\”:

System.out.println (“C:\\companyData\\data.dat”);

The compiler, when faced with two backslashes, compiles it into a single backslash character, and the interpreted string now contains a single backslash at each location. Again, when you begin to interface with the operating system on any computer, which is required when attempting to gain access to any outside resources, it pays to know as much as you can about the OS itself.

Now, back to our file-opening statement. It is the Scanner class that will make sense of the file’s contents. So, let’s consider the next step in setting up our file access mechanism:

new Scanner (new FileInputStream("employeeSales.dat")).

The tooltip for the Scanner class says this: “Constructs a new Scanner that produces values scanned from the specified input stream.” In other words, the FileInputStream is going to connect to the file and grab its stream of characters, then feed those to the Scanner, which will tear them apart and make sense of them. How do we make use of the Scanner? Answer: the same way we retrieved values from the keyboard and placed them into variables. We will make use of the same Scanner methods:

nextInt() for ints

nextLong() for long ints

nextDouble() for doubles

next() for Strings

next().charAt() for characters

The final step in our Scanner-creation statement is to assign the resultant object’s address to the Scanner object called infile. And if there were no problems in all of this process, now infile is a complete interface to the file on disk, and the file is open and ready to go.

**Try/catch block**. It is a fact of programming life that everything does not always go as planned. (And I know I am not telling you any news!) When our program attempts to venture outside itself and touch base with files controlled by the operating system, we have to take a careful approach, lest we fail in our attempts to gain access to them. Accordingly, commercial programs need safeguards to figure out what to do if opening the file is not possible.

What sort of problems might we face? I’m sure you can think of one or more problems that could beset our attempts to get a file open. Perhaps we don’t know the correct filename or simply misspell it. Perhaps the file is not even there at all. Or maybe the file has been opened by some other program and we cannot access it right now. As a result, it’s very important to learn how to anticipate those problems and design the program to handle any problems that may occur.

An extremely common practice is to use what is known as a **try/catch block**. (That is the subject of the several sections listed in your reading schedule at the beginning of Chapter 12.)

The try block is where we execute code that may be risky, and therefore we want to have the capability of catching the sorts of problems that might occur. Then, the catch block contains code that will react to any problems encountered in the try block. Each block is set off by those exact key words, **try** and **catch**. For example, here is a portion of the program we are developing that isolates the use of these two blocks:

try

{

String filename = "employeeSales.dat";

Scanner infile = new Scanner(new FileInputStream(filename));

}

catch (IOException ex)

{

ex.printStackTrace();

}

In the try block we attempt to execute the statement we just finished discussing: the instantiation of the Scanner class object. The riskiest part of that statement is when the FileInputStream constructor executes and attempts to open the file passed in as its argument. If the file cannot be opened, then the operating system generates what is known as an **exception**, an object that describes the problem that must be handled either by your program or the OS itself. The topic of exception handling is a very important one, but for now, we’ll keep it as straightforward as possible.

Our goal is to ensure that our program gets to handle the problem, the exception, and not the OS. If we let the OS handle it, that means one of those ugly-looking dialogs that our program has committed a sin against humanity, and that we deserve to be flogged in the public square. ☺

Handling the problem is the goal of the catch block. The catch block names one exception type that we are looking for, in this case, any sort of IOException, or input/output exception. The Eclipse tooltip window says that this type of exception…“Signals that an I/O exception of some sort has occurred. This class is the general class of exceptions produced by failed or interrupted I/O operations.”

The idea of the syntax, IOException ex, is that if a problem does get generated, the OS first creates the exception object. Then, it looks for a catch block that is engineered to trap the correct type of exception. If ours qualifies, it passes control to that catch block while assigning the address of the object to our variable, ex. Therefore, we will have direct access to the OS’s exception object to see what’s in it.

A common thing to do inside a catch block is to make the call to see the object’s stack trace. The stack trace will tell us exactly where the problem occurred. For example, suppose the filename is not correct, which prevents it from opening. Here is a typical stack trace for such an exception:

java.io.FileNotFoundException: employeeSales.dat (The system cannot find the file specified)

at java.io.FileInputStream.open(Native Method)

at java.io.FileInputStream.<init>(Unknown Source)

at java.io.FileInputStream.<init>(Unknown Source)

at edu.jones.acmepayroll.MainClass.main(MainClass.java:24)

The beginning obviously tells us the sort of exception that occurred, which is contained in a message stored inside the object. And then the last line informs us that it occurred on line 24 of the source code, which “coincidentally” is the line where the call to the FileInputStream attempted to open the file. Later in the course we’ll look at other sorts of exceptions that we need to be on the watch for and how to handle them.

**Complete processing algorithm**. Next, I’ll present a complete main method to process the file, accumulate the total of all sales, and then display the total at the end. Here is the entire main method:

public static void main(String[] args)

{

String salesID;

double salesIn;

double totalSales = 0.0; //Accumulator

try

{

String filename = "employeeSales.dat";

Scanner infile = new Scanner(new FileInputStream(filename));

while (infile.hasNext())

{

//Read one complete record

salesID = infile.next();

salesIn = infile.nextDouble();

//Accumulate total

totalSales += salesIn;

}

//Close the file

infile.close();

System.out.printf("The total sales for the company is $%.2f.\n", totalSales);

}//END try

catch (IOException ex)

{

ex.printStackTrace();

}//END catch

}//END main

Our variable requirements are pretty minimal: two local variables to accept values coming in from the data file, and one accumulator for the total of all sales. Then, inside the try block we declare one more variable for the actual filename and one for the Scanner object. (The only reason I used a separate String for the filename is simply to demonstrate its use.)

Note that the entire file processing part of the algorithm is contained inside the try block. That’s to ensure that if we have any sort of problem with the file at all, the problem can be isolated in that block and be subsequently caught by the catch block. If and when an exception does occur, program execution is halted completely and the OS looks for a suitable catch block to handle the problem.

As long as the file successfully opens, then the while loop takes over. The loop condition takes advantage of a very handy technique offered by the Java class: the ability to look ahead into the file to see if there is anything there to process. The tooltip for the hasNext method says this: “Returns true if this scanner has another token in its input. This method may block while waiting for input to scan. The scanner does not advance past any input.” So, if there is anything more to read, the method returns true (a boolean), otherwise false. That does not guarantee that there is an entire record still out there, but at least we know that there some more to read from the file.

The loop body uses the Scanner class methods to read the next String and the next double from the file stream, and our only processing statement does the required accumulation.

Once the loop quits, we close the file, a very important bookkeeping responsibility for any programmer, and then we display the answer. There are a couple of very important reasons for you to close a file as soon as you are done with it. First, as long as the file is open, you are adding stress to the operating system. Although that may seem pretty trivial on your own computer, it can make a big difference on a commercial server. Never keep resources open that you can give back to the OS. Also, the file may be a limited-use file, meaning that perhaps only one process may open it at a time. That means that as long as you have it open, no one else may open and use it.

If there is an exception generated during our use of the file, the try-catch block makes sure that we get notified and out catch block ends up executing. A common technique is to use the **IOException** object, called here *ex*, to print the stack trace, which was illustrated a couple of pages ago for one file problem. The trace usually makes it a snap to find what line was the offender, and then we can attempt to rectify the problem. The line commented out also illustrates how to display the exception message itself.

Make sure that every part of this algorithm makes sense to you. The file stream class may still be a bit confusing, and that’s all right. But the overall algorithm needs to be clear in your head as you get ready to open and input file data for even more important purposes ahead!

**A handy file-processing tip**: when you’re writing a program to process a data file, I strongly suggest you try doing this. For starters, just code all the file access stuff first. Leave out all processing of the file’s data. Simply read the file from beginning to end to see if your program has any difficulties with it. You might try printing out the value of the last field in each record, nothing else. Check to make sure that this field prints out correctly, all the way to the last record. One of the most common errors encountered with input files involves getting fields all mixed up. Maybe you didn’t declare your variables properly, and when you start inputting data, the variable values get all messed up.

So, files can really be your friends: they’re surprisingly easy to work with. And we’ll make arrays our friends soon! In the next tutorial we will go on to discuss the basics of creating and managing arrays. As you will soon see, reading data files is essential to inputting the vast amount of data you will need to load your arrays.

Let’s finish this discussion by going the “other way”: saving data to an output file on an external drive.

# WRITING DATA FILES

A discussion of data files wouldn’t be complete without considering writing data to an output file. It’s really no more difficult than using input files, though, so let’s dive in for a brief swim.

As an example, let’s suppose we want to create a file that will contain data about a company’s sales staff. We’ll open a brand new file and send several pieces of data about each person to the file. The following will be the record description for the new file:

**RECORD DESCRIPTION (employeeOut.dat):**

|  |  |
| --- | --- |
| lastName | string |
| firstName | string |
| salary | double |

For this sort of file, we will employ the Java PrintWriter class, which will process the data stream sent to a FileWriter class object. Here’s the syntax for instantiating the PrintWriter object:

PrintWriter myPW = new PrintWriter (new FileWriter("employeeData.dat", true));

Just like the Scanner class for file input, which depended on a FileInputStream object for the actual file interface, the PrintWriter class will depend on a FileWriter class object to take care of the nitty-gritty details of the output file. (As before, when you type this sort of code, the IDE will complain about those two terms, PrintWriter and FileWriter. But by mousing over the names, it will suggest that you import their library files, and all will be good once you do that.) The FileWriter constructor will create the new file in the project directory, and it will appear in the Package Explorer window once your program runs the first time. (You can open it by double-clicking on it, and it will open in whatever program you have designated to open a .dat file.)

Note: the FileWriter constructor takes a second argument that indicates whether or not you want to always create a new file or append to an existing file. Pass a value of *true* if you want to append to an existing file (or create a new file if one does not exist), and pass *false* if you always want to create the file from scratch. If you do pass *false*, and the file already exists, the OS will zero out the file and use it as if it were new.

The PrintWriter constructor will run next, finalizing the connection to the newly created file. Like before, we need to embed this in a try block, and use an associated catch block to handle possible exceptions. Fortunately, there is much less that can go wrong with an output file, but it’s important to be ready for all eventualities.

Following is a complete main method to illustrate saving some personnel data to the file. It uses a Scanner class object to get three pieces of data for each person, and then stores them as a single record.

public static void main(String[] args)

{

String lastName, firstName;

double salary;

try

{

//Instantiate a PrintWriter to append to an existing file

PrintWriter myPW = new PrintWriter (new FileWriter("employeeOut.dat", true));

Scanner myScanner = new Scanner (System.in);

//Input first value for lastName, which will be the LCV

System.out.println("Input the employee's last name (or Q to quit): ");

lastName = myScanner.nextLine();

while (!(lastName.equals("Q") || lastName.equals("q")))

{

System.out.println("Input the employee's first name: ");

firstName = myScanner.nextLine();

System.out.println("Input the employee's salary: ");

salary = myScanner.nextDouble();

myScanner.nextLine(); //Eliminate carriage return left in stream after numeric input

//Save one record

myPW.printf ("%s %s %.2f\n", lastName, firstName, salary);

//Input next last name

System.out.println("Input the employee's last name (or Q to quit): ");

lastName = myScanner.nextLine();

}//END while

myPW.close();

myScanner.close();

}//END try

catch (IOException ex)

{

ex.printStackTrace();

}

}//END main method

Here, we are using the lastName variable as the Loop Control Variable (LCV). If the user enters either an upper or a lower case Q, the while loop will terminate. (Be sure you understand why the loop condition will be false for either of those characters.) Note: it made no sense to convert the input to upper or lower case in an effort to simplify the loop condition, since we really don’t want to convert their entire name to that case!

Once we have a complete record in the three variables, the printf statement does the actual file output:

myPW.printf ("%s %s %.2f\n", lastName, firstName, salary);

By using the myPW object, we are sending the output to the PrintWriter object. And that means the entire string will be written to the file as a stream of characters. The formatting we use assures that the record will look correct to us if we later open it up. There are spaces between the fields and a carriage return at the end of each record, just the way we have become accustomed to seeing an input file if we open it as such later on.

And now for one very important last step: remember to close out the file when you are finished working with it. Some operating systems, including some versions of Windows, will hide an output file if it has not been closed by its creator. So, don’t forget that one last step!

That’s it for this first treatise on Java file handling. You should familiarize yourself with all of the code by creating your own project and studying how it works. You can create your own file using any text editor, even Eclipse itself! A short time from now, we will talk further about using an input file to fill up an array. At that time, however, we will be coding the file access algorithm into a class rather than into the main method. Fun times ahead!

### End of Tutorial