**Chapter XVII**

**Input/Output with Sequential Files**

**Chapter XVII Topics**

17.1 Introduction

17.2 Different Types of Files

17.3 Using the File Class

17.4 Files of Character Strings

17.5 Files of Numbers

17.6 A Note About the Scanner Class

17.7 Summary

**17.1 Introduction**

Random Access Memory (*RAM*) is great while it lasts, but it is only temporary. Rebooting the computer, flipping the power switch, tripping over the power cord and enjoying a friendly electrical storm can quickly delete or corrupt the information stored in your computer’s dynamic memory.

Long ago, you became familiar with the solution to this problem. Use external storage to permanently store any important information. Most students learn several lessons about the virtue of saving their work on a regular basis. Common software programs like word processors, spreadsheets, and program languages, like Java, create user data files that are stored externally.

This means that the program you write is in fact a data file of another program. This may seem odd. You probably are comfortable thinking that a word processing document is stored as a data file. However, the programs you have written seem like programs, not data files. Both are true simultaneously. You have written fully functional Java programs. At the same time, the programs you have written were data files of some text editor program, like Notepad or a special Integrated Development Environment (*IDE*) program, JCreator.

Perhaps it has happened to you that the bell rang in the middle of entering data for your program. Or you went to work, to band practice - who knows where - and all the data you entered will need to be entered once more the next time when you return to your computer. Would it not be nice if your program’s data could be stored in the same manner as the program itself?

Program languages in general - and Java specifically - have the ability to create their own data files. This means that data entered by a program user can be stored for future processing. It also means that data only needs to be entered once. In some cases, data never needs to be entered. It is quite possible that your teacher, or somebody else, has created a data file of information. It becomes your job to retrieve that information in your program and process the data. This approach is very convenient and used quite extensively in college computer science classes.

Many computer science courses teach data entry with files quite early, precisely because it is so convenient. I have not used that approach. Personally, I prefer to introduce file handling at a time when I think you have the necessary tools to handle the topic properly. You will find that input/output processes in Java are hardly a trivial task.

|  |
| --- |
| **Programs and Data Files Notes** |
| Data information processed by word processing, spreadsheet  and similar application programs are stored as data files.  Java programs, written with a text editor or some Integrated  Development Environment (*IDE*), are also data files.  Programs have the capability to create and retrieve their  own data files. This means that data entered in any one of  your programs can be saved for later use.  Programs can be tested with existing data files for greater  efficiency. |

This chapter will only present a small quantity of all the many classes that Java has created to process the input and output of data streams and files. Pick up any complete reference manual on Java and you will be amazed at the variety of ways that Java can manage data. There are classes for every conceivable input/output data processing situation. Any attempt to even give a basic introduction to each one of these classes would become a lesson in overwhelming frustration. The information provided in this chapter will give you a comfortable and practical start in managing data that is transferred between the computer and external storage.

|  |
| --- |
| **AP Computer Science Examination Alert** |
| The material presented in this chapter will not be tested on the AP Computer Science Examination, precisely because of the large variety of approaches that can be used to achieve similar input/output goals. |

The most fundamental justification for using a computer is to process information or data. Frequently, data need to be transferred from some external storage to internal memory or newly processed internal data need to be stored externally for future processing needs. The word *data*is tossed around in computer science in general and it certainly is mentioned every other sentence already in this chapter. What exactly is data, and how does it fit in with this file business? Computer data forms an organized hierarchy of information. Data starts at the very humble *bit* and works up the ranks to *byte*, *field*, *record* and *file*. Let's look at each one of these stages.

|  |  |
| --- | --- |
| **Data Organization** | |
| **bit** | This is the fundamental building block of all computer information, a binary digit, which stores only **1** or **0**. |
| **byte** | One **byte** equals **8 bits**.  One character can be stored in one byte, using ASCII code.  One character is stored in two bytes using Unicode. |
| **field** | A **field** is one specific unit of information of one data type, such as **size**, **age**, **name**, **date**, etc. |
| **record** | A record consists of a set of fields for some specific purpose, such as a **student** record, a **medical** record, an **employee** record, an **airline passenger** record, etc. |
| **file** | A **file** is a sequence of records of the same type. |

The transfer of data between internal memory and external storage is quite a complicated process. Modern languages like C++ and Java use *streams* to handle data transfer. A *stream* is a set of bytes or characters. In the old days there was no distinction between bytes and characters. Now that *Unicode*, with its *two-byte* characters is in use, there is a distinction. Think of a stream as a pipeline that contains a neatly ordered set of data. Streams can go in two directions for input and for output. Input streams can come from the keyboard, from an external storage source, or some GUI mouse-click. Output streams can go to the monitor, an external storage location or the printer. Streams allow consistent input/output processing for many different types of sources and destinations.

**17.2 Different Types of Files**

It is very important to understand the difference between a *file*and a *file data structure*. It is easier to explain what a file is. Working on a computer means working with files. Files contain information used by some application program. So what is a file? You just saw this data hierarchy, and it appears that a file is a sequence of related records. From the computer's point of view a file only holds information. The programmer organizes the information in files, in records of fields and so on. A file really is nothing more than a sequence of information stored on a disk. The manner in which the data is accessed varies but the file itself is a bunch of information coded on a disk or other storage device.

|  |
| --- |
| **File Definition** |
| A *file*is a sequence of information stored as bytes on a  disk, tape, CD or other external storage device. |

Understanding files is assisted by a brief history lesson of external storage. Early storage devices used long tapes on large disks. Accessing information on such tapes was only possible in sequence. Reading or writing information with tape storage required sequential access. Most students are familiar with this process. Pop an audiotape in a player and try to play the third song. It is necessary to listen to the first two songs first or fast forward to your desired selection. Either way, it is not possible to access the requested song directly. Access on a tape is sequential. Early computers with tape storage developed programs that could only use sequential file data access.

With time, external storage switched to disks and CDs. Such devices allow direct access of certain files. This type of access is called *random access*. You can randomly access file information. This type of access is similar to an old record album player or CD player. It is possible to jump directly to the third song or any other desired song on the album or CD.

Random access of files might be handled in the second computer science course or some other future computer science course. In this first file introduction we will only concentrate on sequential files. After you have worked with sequential files, it will be a comfortable transition to add the additional processing required to manage random access files.

|  |
| --- |
| **Sequential Access and Random Access** |
| Files can have sequential access or random access.  *Sequential access*files allow data access only in the  sequence that the data is stored in the file.  *Random access*files allow data access in any random  pattern, regardless of how the data is stored. |

**17.3 Using the File Class**

Before we generate too much excitement about transferring between internal and external computer locations, we need to stop and realize that major problems will result when desired file locations cannot be found. Suppose that you try to retrieve a set of data. This data is stored in a file with a name and the file is stored in some directory. Any program that needs to access the data file needs to know the correct file name and the correct file location. If the file name is typed incorrectly, if the file name is unknown, if the file is deleted, if the program looks in the wrong directory, your data processing will come to a screeching halt. First and foremost, you need to know that the data transfer you are about to process has the possibility of success. Such information can be done very nicely with the **File** class. The **File** class has a set of very convenient methods that assist in file information without actually processing any data.

Program **Java1701.java**, in figure 17.1, checks to see if two different files exist. The program first checks for the "**qwerty.dat"** file and then the "**Java1701.dat"** file. The "**qwerty.dat"** file name is strictly fictitious and is meant to test the program. Two **File** methods are used. The **exists** method returns **true** if the specified file exists and the **getName** method returns the name of the external file.

## Figure 17.1

|  |
| --- |
| // Java1701.java// This program demonstrates how to check if an external text file exists// using the <getName> and <exists> methods of the <File> class.import java.io.\*;public class Java1701{  public static void main (String args[])  {  File file1 = new File("qwerty.dat"); // #1  System.out.print(file1.getName()); // #2  if (file1.exists()) // #3  System.out.println(" exists.");  else  System.out.println(" does not exist.");    File file2 = new File("Java1701.dat");  System.out.print(file2.getName());  if (file2.exists())  System.out.println(" exists.\n");  else  System.out.println(" does not exist.\n");  }  } |

|  |
| --- |
| **Java1701.java Output**  qwerty.dat does not exist.  Java1701.dat exists. |

Program **Java1701.java** has three significant statements that you have not seen in any earlier program. These three statements are numbered and will be repeated below for additional explanation.

**File file1 = new File("qwerty.dat"); // #1**

Statement **#1** instantiates a new **File** object, called **file1**,with the **"qwerty.dat"** parameter. This statement associates the internal File object with the external "qwerty.dat" file name.

**System.out.print(file1.getName()); // #2**

Statement **#2** returns the external file name associated with the **file1** object.

**if (file1.exists()) // #3**

Statement **#3** returns **true** if **"qwerty.dat"** exists and returns **false** otherwise.

|  |
| --- |
| **File Methods getName and exists** |
| **File f = new File("test.dat");**  **String s = f.getName();**  Method **getName** returns file name **"test.dat"**. |
| **if (f.exists())**  Method **exists** returns **true** if **"test.dat"** exists, and returns **false** if the file does not exist. |

Program **Java1702.java**, in figure 17.2, demonstrates many methods of the useful **File** class. The **getName** and **exists** methods of the previous program are repeated. Additionally, there is the **length** method, which returns the size of the file, the **getAbsolutePath** method, which returns the entire file path, the **canRead** method, which returns true if the file is readable and the **canWrite** method, if the file is writable. What is this **readable** and **writeable** stuff? It is possible to block *read privileges* to a file, but most files are readable. Many files however are **read-only**, which prevents any alteration of the file.

## Figure 17.2

|  |
| --- |
| // Java1702.java// This program demonstrates the <getName>, <exist>, <length>, // <getAbsolutePath>, <canRead> and <canWrite> methods of the File class.import java.io.\*;public class Java1702{ public static void main (String args[])  {  File f = new File("Java1702.dat");  System.out.println("File name: " + f.getName());  System.out.println("Does file exist: " + f.exists());  System.out.println("File size: " + f.length());  System.out.println("Complete file path: " + f.getAbsolutePath());  System.out.println("File is readable: " + f.canRead());  System.out.println("File is writeable: " + f.canWrite());  System.out.println();  }  } |

**Figure 17.2 Continued**

|  |
| --- |
| **Java1702.java Output**  File name: Java1702.dat  Does file exist: true  File size: 148  Complete file path: C:\Java2002\JavProgs\Progs17\Java1702.dat  File is readable: true  File is writeable: true |

|  |
| --- |
| **File Methods length, getAbsolutePath, canRead & canWrite** |
| **int n = f.length();**  Method **length** returns the size of **f**. |
| **String p = f.getAbsolutePath();**  Method **getAbsolutePath** returns the complete path from root to the file directory of **f**. |
| **boolean canIt = f.canRead();**  Method **canRead** returns true if **f** is readable. |
| **boolean canIt = f.canWrite();**  Method **canWrite** returns true if **f** is writeable. |

The **file** class also has some methods to create and delete files. The **delete** method removes an existing file. The **createNewFile** method makes a new file. There is little confusion about the **delete** method. A file exists with a lot of data, little data or no data, it does not matter because **delete** removes the entire file, data and all.

The **createNewFile** method does not transfer any information. It creates an empty shell ready to store information. Sometimes it is useful to create a variety of files for some future purpose. You may have files for different conference participants, and until the first participant registers, the file is empty. Program **Java1703.java**, in figure 17.3, plays around with these two methods and proves that files are in fact created and deleted.

## Figure 17.3

|  |
| --- |
| // Java1703.java  // This program demonstrates creating and deleting files with the  // <delete> and <createNewFile> methods.  import java.io.\*;  public class Java1703  {  public static void main (String args[]) throws IOException  {  System.out.println("\nJava1703.java\n");  File f = new File("Java1703.dat");  System.out.println("Before <create> File exists: " + f.exists());  f.createNewFile();  System.out.println("After <create> File exists: " + f.exists());  f.delete();  System.out.println("After <delete> File exists: " + f.exists());  System.out.println();  }  } |

|  |
| --- |
| **Java1703.java Output**  Java1703.java  Before <create> File exists: false  After <create> File exists: true  After <delete> File exists: false |

|  |
| --- |
| **File Methods delete and createNewFile** |
| **f.delete();**  Method **delete** removes the external file associated with **f**. |
| **f.createNewFile();**  Method **createNewFile** creates a new, but empty file. |

**17.4 Files of Character Strings**

Java has a very powerful **String** class and you saw how many methods are available to manipulate **String** objects, in the previous chapter. Java also has some **Writer** and **Reader** classes that are custom ordered for working with file string manipulation. I have mentioned before that Java has developed a tremendous number of different classes to process every conceivable data stream of information. We are not going to learn, or even look at, all these different possibilities. What is it that we really like to do to make our life comfortable with file processing? How about writing and reading one complete line of text all at once. Would it not be simpler to process lines rather than characters?

We can achieve this by combining several classes. The **FileWriter** and **FileReader** classes create objects that handle character based streams. This is a good start in making our life simpler. We continue with the **BufferedWriter** and the **BufferedReader** classes. These classes contain methods that allow working with *line-oriented* files. Program **Java1704.java**, in figure 17.4, starts gently by transferring one complete line of text to an external file. You will not see any evidence that data is converted between bytes and characters. Figure 17.5 shows the view of the actual **"Java1706.dat"** contents, proving that the entire line is transferred to external storage.

## Figure 17.4

|  |
| --- |
| // Java1704.java// This program demonstrates how to create textfiles with the <FileWriter> class,// <BufferedWriter> class and the <write> method.  import java.io.\*;  public class Java1704  {  public static void main (String args[]) throws IOException  {  FileWriter outFile = new FileWriter("Java1704.dat"); // #1  BufferedWriter outStream = new BufferedWriter(outFile); // #2  String outString = "Too bad Java has so few I/O classes"; // #3  outStream.write(outString); // #4  outStream.close(); // #5  System.out.println("Java1704.dat is created\n"); // #6  }  } |

**Figure 17.5**

|  |
| --- |
| **Contents of "Java1704.dat" file**  Too bad Java has so few I/O classes |

**Figure 17.4 Continued**

|  |
| --- |
| **Java1704.java Output**  Java1704.dat is created |

You have heard the term *buffer* used in various previous chapters. A *buffer* is a temporary memory location that handles the transfer between internal memory and peripheral computer devices, such as printers, keyboard, and disk drives. The transfer of each single byte to external storage is an inefficient process. Internal data can wait in the buffer until a reasonable amount of data collects in the buffer and then the transfer is made. If you need to take ten books to the library, do you make ten trips with one book or do you make one trip with ten books? The **BufferedWriter** and **BufferedReader** classes fill one line of information. When the *end-of-line* character is detected everything in the buffer is transferred.

**FileWriter outFile = new FileWriter("Java1704.dat"); // #1**

A **FileWriter** character-based object **outFile** is constructed and associated with external file **"Java1704.dat"**.

**BufferedWriter outStream = new BufferedWriter(OutFile); // #2**

A **BufferedWriter** object, **outStream**, is constructed and associated with the **outFile** object.

**String outString = "Too bad Java has so few I/O classes"; // #3**

One character string object of information is initialized.

**outStream.write(outString); // #4**

Method **write** takes the **outString** object and transfers its contents to external storage.

**outStream.close(); // #5**

The file output stream business is concluded with a call to the **close** method.

**System.out.println("Java1704.dat is created"); // #6**

The result of executing program **Java1704.java** is now visible on the monitor.

Program **Java1705.java**, in figure 17.6, is a repeat of the previous program. The same classes, methods and string information are processed. The only difference is that the **FileWriter** and **BufferedWriter** classes are "wrapped" into a single statement.

## Figure 17.6

|  |
| --- |
| // Java1705.java// This program is identical to Java1704.java with the <FileWriter> class// and <BufferedWriter> class statements combined in one statement.import java.io.\*;public class Java1705  {  public static void main (String args[]) throws IOException  {  BufferedWriter outStream = new BufferedWriter(new FileWriter("Java1705.dat"));  String outString = "Too bad Java has so few I/O classes";  outStream.write(outString);  outStream.close();  System.out.println("Java1705.dat is created\n");  }  } |

|  |
| --- |
| **Java1705.java Output**  Java1705.dat is created |

|  |
| --- |
| **FileWriter and BufferedWriter Classes** |
| The **FileWriter** class is used to transfer character-oriented data from internal memory to an external file. The **BufferedWriter** class manages line-oriented data. Together these two classes enable you to transfer line-oriented character strings.  **FileWriter outFile = new FileWriter("Java1705.dat");**  **BufferedWriter outStream = new BufferedWriter(outFile);** |

Figure 17.7 displays the two statements used by program **Java1704.java** to create two objects used to transfer data. The second display accomplishes the same goal in one statement apparently with one - **outStream** - object. The parameter used to create the **outStream** object is **new FileWriter("Java1705.dat")**, which is an object. This is an unusual object, because there is no object identifier in sight. Such an object is called an *anonymous object* in computer science and you have seen this in various GridWorld programs.

**Figure 17.7**

|  |
| --- |
| FileWriter outFile = new FileWriter("Java1704.dat");  BufferedWriter outStream = new BufferedWriter(OutFile); |
| BufferedWriter outStream = new BufferedWriter(new FileWriter("Java1705.dat")); |

|  |
| --- |
| **Anonymous Objects** |
| In **Java1705.java** the following single statement is used.    **BufferedWriter outStream = new BufferedWriter(new FileWriter("Java1705.dat"));**  Note that the **previous outFile** object is no longer used, but there is a **new BufferedWriter** object, which requires an object.  An object, without a name, is created by  **new FileWriter("Java1705.dat").** This object without an identifying name is called an *anonymous* object. |

## With program Java1706.java, shown in figure 17.8, we switch from Writer classes to reader classes and retrieve the one-line "Java1705.dat" text file. Pay particular attention to the readLine method. This is the method that makes it possible to retrieve an entire line of characters, including the end-of-line character.

## Figure 17.8

|  |
| --- |
| // Java1706.java  // This program demonstrates how to retrieve textfiles with the <FileReader> class,  // <BufferedReader> class and the <read> method. This program will retrieve  // the textfiles created by the Java1705.java.  import java.io.\*;  public class Java1706  {  public static void main (String args[]) throws IOException  {  FileReader inFile = new FileReader("Java1705.dat"); // #1  BufferedReader inStream = new BufferedReader(inFile); // #2  String inString = inStream.readLine(); // #3  System.out.println(inString); // #4  inStream.close(); // #5  System.out.println();  }  } |

|  |
| --- |
| **Java1706.java Output**  Too bad Java has so few I/O classes |

**FileReader inFile = new FileReader("Java1705.dat"); // #1**

A **FileReader** character-based object **inFile** is constructed and associated with external file **"Java1705.dat"**.

**BufferedReader inStream = new BufferedReader(inFile); // #2**

A **BufferedReader** object, **inStream**, is constructed and associated with the **inFile** object.

**String inString = inStream.readLine(); // #3**

Method **readLine** takes one entire line of characters and transfers its contents from external storage to internal memory.

**System.out.println(inString); // #4**

The string retrieved by **readLine** is displayed.

**inStream.close(); // #5**

The file input stream business is concluded with a call to the **close** method.

The significance of using the **BufferedWriter** and **BufferedReader** classes can easily be lost by the program examples that you just saw. The actual transfer of data consisted of just a single line of characters. You just saw how the **readLine** method can transfer an entire line of characters, including the **end-of-line** character. This is great, but how does this fancy character manage to find itself at the conclusion of each line? This job is accomplished by the **newLine** method of the **BufferedWriter** class. The **newLine** method was not used in any previous program because only a single line was transferred. In program **Java1707.java**, shown in figure 17.9, four lines of data are transferred.

## Figure 17.9

|  |
| --- |
| // Java1707.java// This program demonstrates how to create a multi-line textfile separated with the <newLine> method.import java.io.\*;public class Java1707{ public static void main (String args[]) throws IOException  {  BufferedWriter outStream =  new BufferedWriter(new FileWriter("Java1707.dat")); // #1  outStream.write("The quick brown "); // #2  outStream.write("fox jumps over the lazy dog"); // #3  outStream.newLine(); // #4  outStream.write("on Sundays only,"); // #5  outStream.newLine(); // #6  outStream.write("unless it rains."); // #7  outStream.close(); // #8  System.out.println("Java1707.dat is created\n"); // #9  }  } |

**Figure 17.9 Continued**

|  |
| --- |
| **Java1707.java Output**  Java1707.dat is created |

Figure 17.10 shows the contents of the Java1707.dat file, created by the Java1707.java program. Did you expect the file to appear like this?

**Figure 17.10**

|  |
| --- |
| **Contents of "Java1707.dat" file**  The quick brown fox jumps over the lazy dogon Sundays only,unless it rains. |

Note that the **write** method transfers a string of characters very nicely to the external file, but not an **end-of-line** character. The *The quick brown* string is immediately followed by the *fox jumps over the lazy dog* string stored externally on the same line. It is precisely the **newLine** method call that comes to the rescue and sees to it that any following statements will start on a *new line*.

**BufferedWriter outStream =**

**new BufferedWriter(new FileWriter("Java1707.dat")); // #1**

An **outStream** object is instantiated ready to transfer some characters from internal memory to the external file **"Java1707.dat"**.

**outStream.write("The quick brown "); // #2**

Method **write** transfers the *The quick brown* string to the external text file.

**outStream.write("fox jumps over the lazy dog"); // #3**

Method **write** transfers the *fox jumps over the lazy dog* string to the external text file, and appends this string immediately next to the previously transferred string.

**outStream.newLine(); // #4**

Method **newLine** adds an **end-of-line** character to the external file. The result is that the next data will be stored on a new line in the text file.

**outStream.write("on Sundays only,"); // #5**

Method **write** transfers the *on Sundays only* string to the external file. The string is stored immediately below the previously transferred string.

**outStream.newLine(); // #6**

Method **newLine** adds an **end-of-line** character to the external file. The result is that the next data will be stored on a new line in the text file.

**outStream.write("unless it rains."); // #7**

Method **write** transfer the *unless it rains* string to the external file. The string is stored immediately below the previously transferred string.

**outStream.close(); // #8**

The file output stream business is concluded with a call to the **close** method.

**System.out.println("Java1707.dat is created"); // #9**

The result of executing program **Java1707.java** is now visible on the monitor.

|  |
| --- |
| **BufferedWriter Methods write, newLine and close** |
| **outStream.write("Grace Hopper is my hero");**  Method **write** transfers character stream data from internal  memory to external storage. |
| **outStream.newLine();**  Method **newLine** adds an **end-of-line** character to the external file. The result is that the next data will be stored on a new line in the text file. |
| **outStream.close();**  Method **close** concludes the output stream processing. |

Something interesting needs to be considered about file processing. When have you finished reading the entire file? This is easy with one-line text file. A single **readLine** method call and the job is done. It is also simple if you know the actual contents of the data file, but that is not often the case. Besides, programmers have no knowledge of future data file contents and program code must be more general, to be useful. What comes to the rescue is the humble **null** keyword. Use a conditional while **loop** and compare the result of the file transfer with **null**. If **null** is **true**, then transfer is finished and the loop stops.

Program **Java1708.java**, in figure 17.11, is in charge of retrieving the multi-line text file created by the previous program. You will see that the **readLine** method call is part of the **while** loop condition. The result is a continuous repetition of *one-line* data transfers until the entire textfile is transferred to internal memory.

## Figure 17.11

|  |
| --- |
| // Java1708.java  // This program reads in the entire Java1707.dat or any other text file,  // line by line. Additionally a <Scanner> object to enter a file name  // during program execution.  import java.io.\*;  import java.util.Scanner;  public class Java1708  {  public static void main (String args[]) throws IOException  {  System.out.println("\nJava1708.java\n");  Scanner keyboard = new Scanner(System.in);  System.out.print("Enter an external file name ===>> ");  File inFile = new File(keyboard.nextLine());  if (inFile.exists())  {  BufferedReader inStream =  new BufferedReader(new FileReader(inFile)); // #1  String inString; // #2  while((inString = inStream.readLine()) != null) // #3  System.out.println(inString); // #4  inStream.close();  }  else  {  System.out.println("Specified file does not exist.");  System.out.println("Program aborted.");  }  System.out.println();  }  } |

|  |
| --- |
| **Java1708.java Output #1**  Java1708.java  Enter an external file name ===>> Java1707.dat  The quick brown fox jumps over the lazy dog  on Sundays only,  unless it rains. |

This program has the ability to display the contents of any text file. Remember that .java files are themselves text files. Does that mean this program can display other Java programs, or even itself? The answer is “Yes” as the next output demonstrates.

## Figure 17.11 Continued

|  |
| --- |
| **Java1708.java Output #2**  Java1708.java  Enter an external file name ===>> Java1708.java  // Java1708.java  // This program reads in the entire Java1707.dat or any other text file,  // line by line. Additionally a <Scanner> object to enter a file name  // during program execution.  import java.io.\*;  import java.util.Scanner;  public class Java1708  {  public static void main (String args[]) throws IOException  {  System.out.println("\nJava1708.java\n");  Scanner keyboard = new Scanner(System.in);  System.out.print("Enter an external file name ===>> ");  File inFile = new File(keyboard.nextLine());  if (inFile.exists())  {  BufferedReader inStream =  new BufferedReader(new FileReader(inFile)); // #1  String inString; // #2  while((inString = inStream.readLine()) != null) // #3  System.out.println(inString); // #4  inStream.close();  }  else  {  System.out.println("Specified file does not exist.");  System.out.println("Program aborted.");  }  System.out.println();  }  } |

The final example shows what happens when an erroneous file name is entered. If the special provision to check if the file exists was not in the program, this program would crash with this input.

## Figure 17.11 Continued

|  |
| --- |
| **Java1708.java Output #3**  Java1708.java  Enter an external file name ===>> qwerty.dat  Specified file does not exist.  Program aborted. |

**BufferedReader inStream =**

**new BufferedReader(inFile); // #1**

An **inStream** object is instantiated ready to transfer some characters from the external text file stored in **inFile**, indicated at the command prompt, to internal memory.

**String inString; // #2**

Object **inString** is constructed to stores individual lines retrieved from the external text file.

**while((inString = inStream.readLine()) != null) // #3**

This is an important statement. A call to the **readLine** method transfers an entire line of characters from the external file and stores the line in the **inString** object. The **while** loop condition remains **true** as long as there are characters to transfer. When the **null** character is reached - meaning file is empty - the loop stops.

**System.out.println(inString); // #4**

Each individual line from the text file is stored in **inString** and then displayed by the **println** method.

|  |
| --- |
| **FileReader and BufferedReader Classes** |
| The **FileReader** class is used to transfer character-oriented data from an external file to internal memory. The **BufferedReader** class manages line-oriented data. Together these two classes enable you to transfer line-oriented character strings.  **FileReader inFile = new FileReader("Java1707.dat");**  **BufferedReader inStream = new BufferedReader(inFile);**  These two statements can be "wrapped" together into a single statement with the same result.    **BufferedReader inStream = new BufferedReader(new FileReader(args[0]));** |

|  |
| --- |
| **BufferedReader Methods write, readLine and close** |
| **inString = inStream.readLine();**    Method **readLine** transfers one complete line of characters, including the **end-of-line** character, from external storage to internal memory. |
| **inStream.close();**  Method **close** concludes the input stream processing. |

**17.5 Files of Numbers**

You now should have a pretty decent grasp on file structures, at least on text file structures that store characters. Can you store something besides characters? Actually, you can store a bunch of different data types using many different file classes. Frankly, that is precisely the complexity that we have no interest in exploring right now. There does remain the important issue of numerical data. It is true that numerical digit characters are in fact characters and can be stored in such a manner. The problem is that numerical characters cannot be processed for any mathematical computation. What we need is to be a little tricky. There are clever methods that convert numbers to strings and strings to numbers. Anytime we need to work with numbers then let us store the value internally in integers and doubles. When the time comes to store the numerical data, convert it to strings, and transfer it to a text file. If you need to work with the data at a later data, reverse the process and convert the characters back to numbers. Program **Java1709.java**, in figure 17.12, gets the balls rolling by generating ten integers in the **[1000..9999]** range. The numbers are transferred to the "**Java1709.dat"** file, but first each number is converted with the **String.valueOf** method. Besides the random number generation and the conversion, it is business as usual in the file department.

## Figure 17.12

|  |
| --- |
| // Java1709.java// This program generates a textfile of 10 random integers.// Each integer is converted with the <String.valueOf> method // to a string before the number is transferred to the OutputStream.  import java.io.\*;  import java.util.Random;  public class Java1709  {  public static void main (String args[]) throws IOException  {  Random rand = new Random(12345);  BufferedWriter outStream =  new BufferedWriter(new FileWriter("Java1709.dat")); // #1  int rndInt; // #2  for (int k = 1; k <= 10; k++) // #3  {  rndInt = rand.nextInt(9000) + 1000; // #4  outStream.write(String.valueOf(rndInt)); // #5  outStream.newLine(); // #6  }  outStream.close(); // #7  System.out.println("Java1709.dat is created\n"); // #8  }  } |

|  |
| --- |
| **Java1709.java Output**  Java1709.dat is created |

**Figure 17.13**

|  |
| --- |
| **Contents of "Java1709.dat" file**  6251  6080  9241  1828  4055  2084  2375  9802  2501  5389 |

The contents of the **"Java1709.dat"**, in figure 17.13,display a lovely set of random numbers. Are they numbers or are they characters? They definitely are characters. The contents of the file are viewed with Notepad, and Notepad is strictly a text editor capable of only opening up text files of characters. The secret is in converting between numbers and strings. Check out the program line by line for better clarification of the different stages.

**BufferedWriter outStream =**

**new BufferedWriter(new FileWriter("Java1709.dat")); // #1**

An **outStream** object is instantiated ready to transfer some characters from internal memory to the external file **"Java1709.dat"**.

**int rndInt; // #2**

This integer variable stores random numbers in the **[1000..9999]** range.

**for (int k = 1; k <= 10; k++) // #3**

A fixed loop that generates a random integer, converts the integer to a string, and transfers the converted number to an external file, ten times.

**rndInt = rand.nextInt(9000) + 1000; // #4**

This statement generates a random integer in the **[1000..9999]** range.

**outStream.write(String.valueOf(rndInt)); // #5**

Two steps are performed in one statement. First the integer value stored in **rndInt** is converted to a string with the **String.valueOf** method. Second, the converted string is transferred to external storage with the **write** method.

**outStream.newLine(); // #6**

The **newLine** method adds an **end-of-line** character after each numeric string transfer to the external file.

**outStream.close(); // #7**

The file output stream business is concluded with a call to the **close** method.

**System.out.println("Java1709.dat is created"); // #8**

The result of executing program **Java1709.java** is now visible on the monitor.

|  |
| --- |
| **String Method valueOf** |
| **String S = String.valueOf(intNumber);**  Static method **valueOf** converts **intNumber** to a String object.  This is not a file class method, but you need to use this method to store numerical values. |

Now comes the real test. Program **Java1709.java** generated ten random numbers, converted them to strings, and stored them externally in **"Java1709.dat"**. It will be the job of program **Java1710.java**, in figure 17.14, to retrieve the character numbers file and convert the numerical characters back to numbers. Our test to determine if they are really integers once again is to compute, and display, the **Sum** of the ten random integers.

## Figure 17.14

|  |
| --- |
| // Java1710.java// This program retrieves the random integer textfile created by program  // Java1709.java. The stored character strings are converted back to  // integers. The integer value of the integers is computed and displayed  // to prove that the values are in fact integers.  import java.io.\*;  public class Java1710  {  public static void main (String args[]) throws IOException  {  BufferedReader inStream =  new BufferedReader(new FileReader("Java1709.dat")); // #1  String inString; // #2  int rndInt; // #3  int sum = 0; // #4  while((inString = inStream.readLine()) != null) // #5  {  System.out.println(inString); // #6  rndInt = Integer.parseInt(inString); // #7  sum += rndInt; // #8  }  inStream.close(); // #9  System.out.println("sum equals " + sum); // #10  System.out.println();  }  } |

|  |
| --- |
| **Java1710.java Output**  6251  6080  9241  1828  4055  2084  2375  9802  2501  5389  Sum equals 49606 |

The output display proves that the ten numbers were treated like integers and performed an arithmetic operation. Strings do not add numbers. Any attempt to add ten strings would result in a very long set of *concatenated* numerical characters, and not a mathematical sum. Let us check out each step.

**BufferedReader inStream = new BufferedReader(new FileReader("Java1709.dat")); // #1**

An **inStream** object is instantiated ready to transfer some characters from the external text file stored in **args[0]**, indicated at the command prompt, to internal memory.

**String inString; // #2**

The **inString** objectstores data transferred from the external file.

**int rndInt; // #3**

**RndInt** stores the integer value after the file string is converted

**int sum = 0; // #4**

This statement initializes variable **sum** to zero, and will be used to stores the subtotal of each converted string to integer value.

**while((inString = inStream.readLine()) != null) // #5** A call to the **readLine** method transfers an entire line of characters from the external file and stores the line in the **inString** object. The **while** loop condition remains **true** as long as there are characters to transfer. When the **null** character is reached the loop stops.

**System.out.println(inString); // #6**

This statement prints the text file information after it is transferred to memory.

**rndInt = Integer.parseInt(inString); // #7**

The **Integer.parseInt** method convert the **InString** value to an integer.

**sum += rndInt; // #8**

**sum** accumulates a running subtotal for each converted integer.

**inStream.close(); // #9**

The file input stream business is concluded with a call to the **close** method.

**System.out.println("sum equals " + sum); // #10**

The value of **Sum** is displayed on the monitor.

|  |
| --- |
| **Integer Method parseInt** |
| **rndInt = Integer.parseInt(inString);**  Static method **parseInt** converts **inString** to an integer value.  This is not a file class method, but you need to use this method to convert numerical character values back to integers. |

**17.6 A Note About the Scanner Class**

The **Scanner** class was introduced with Java JDK version **1.5.0**. The **Scanner** classsimplified keyboard input and was introduced back in Chapter V to allow program interaction. Technically, methods of the **Scanner** class are file processing methods. You have only used **nextLine**, **nextInt** and **nextDouble** to enter strings, integers and real numbers from the keyboard, but realize that the keyboard is considered an external file.

These same methods can be used with external files, such as the ones used in this chapter. The problem is that the **Scanner** class only provides methods for file input and not for file creation. The system of using two file objects with a combination of **FileWriter** and **BufferedWriter** will still need to be used. There are also added complications using methods **nextInt** and **nextDouble** since these methods require integers and real numbers. The approach in this chapter has been to use only text files. Using the **Scanner** class for external data file input will require additional processing to store integers and real numbers.

The **Scanner** class does simplify keyboard input, but it is not really simpler for the file handling introduced in this chapter. Please keep in mind that file processing is a very involved subject in Java and this chapter has introduced a small subset of file classes. This small group of classes provides a very practical functionality for handling for most file handling situations.

For the same reason, it is also possible to use the combination of the **BufferedReader** and **InputStreamReader** classes for keyboard input. This actually is how *Exposure Java 2004 Edition* presented keyboard input before we had the **Scanner** class. Two more programs will be shown. The first will review input with the **Scanner** class.

## Figure 28.15

|  |
| --- |
| // Java1711.java  // This program reviews some of the methods used by the <Scanner> class.  import java.util.Scanner;  public class Java1711  {  public static void main (String args[])  {  System.out.println("\nJava1711.java\n");  Scanner input = new Scanner(System.in);    String name;  int age;  double gpa;  System.out.print("Enter Name ===>> ");  name = input.nextLine();  System.out.print("Enter Age ===>> ");  age = input.nextInt();  System.out.print("Enter GPA ===>> ");  gpa = input.nextDouble();  System.out.println();  System.out.println("Name: " + name);  System.out.println("Age: " + age);  System.out.println("GPA: " + gpa);  System.out.println();  }  } |

|  |
| --- |
| **Java1711.java Output**  Java1711.java  Enter Name ===>> Jessica Schram  Enter Age ===>> 8  Enter GPA ===>> 4.0  Name: Jessica Schram  Age: 8  GPA: 4.0 |

The final program will have the same output as the previous one. This will demonstrate how to use the combination of the **BufferedReader** and **InputStreamReader** classes for keyboard input. Note that this form of input (as with most forms of input in Java) only can enter a **String**. The **parseInt** and **parseDouble** methods are necessary in this program for conversion.

## Figure 28.16

|  |
| --- |
| // Java1712.java  // This program shows how BufferedReader and InputStreamReader can be used for  // keyboard input.  import java.io.\*;  public class Java1712  {  public static void main (String args[]) throws IOException  {  System.out.println("\nJava1712.java\n");  BufferedReader input = new BufferedReader(new InputStreamReader(System.in));    String name;  int age;  double gpa;  System.out.print("Enter Name ===>> ");  name = input.readLine();  System.out.print("Enter Age ===>> ");  age = Integer.parseInt(input.readLine());  System.out.print("Enter GPA ===>> ");  gpa = Double.parseDouble(input.readLine());  System.out.println();  System.out.println("Name: " + name);  System.out.println("Age: " + age);  System.out.println("GPA: " + gpa);  System.out.println();  }  } |

|  |
| --- |
| **Java1712.java Output**  Java1712.java  Enter Name ===>> Jessica Schram  Enter Age ===>> 8  Enter GPA ===>> 4.0  Name: Jessica Schram  Age: 8  GPA: 4.0 |

**17.7 Summary**

Data information processed by word processing, spreadsheet, or other application programs is stored as data files.

Java programs written with a text editor or some Integrated Development Environment (*IDE*) are also data files.

Programs have the capability to create and retrieve their own data files. This means that data entered in any one of your programs can be saved for later use.

Programs can be tested with existing data files for greater efficiency.

The material presented in this chapter will not be tested on the AP Computer Science Examination, precisely because of the large variety of approaches that can be used to achieve similar input/output goals.

A **file** is a sequence of information stored as bytes on a disk, tape, CD or other external storage device.

Files can have sequential access or random access. *Sequential access* files allow data access only in the sequence that the data is stored in the file. *Random access*files allow data access in any random pattern, regardless of how the data is stored.

**File** method **getName** returns the file name of an external.

**File** method **exists** returns **true** if a file exists, and returns **false** if the file does not exists.

**File** method **length** returns the size of a file.

**File** method **getAbsolutePath** returns the complete path from root to a file.

**File** method **canRead** returns true if a file is readable.

**File** method **canWrite** returns true if a fileis writeable.

**File** method **delete** removes an external file.

**File** method **createNewFile** creates a new, but empty file.

The **FileOutputStream** class is used to transfer byte-oriented data from internal memory to an external file. Objects can be constructed with a **String** file name or with a **File** class object.

**FileOutputStream output = new FileOutputStream("test.dat");**

**File f = new File("Java1704.dat");**

**FileOutputStream output = new FileOutputStream(f);**

**FileOutputStream** method **write** transfers of data from internal memory to external storage.

**FileOutputStream** method **close** closes the output stream file. Every file class has a **close** method that stops the stream of data flow.

**OutStream.close();**

The **FileInputStream** class is used to transfer byte-oriented data from an external file to internal memory. Objects can be constructed with a **String** file name or with a **File** class object.

**FileInputStream input = new FileInputStream("test.dat");**

**File f = new File("Java1704.dat");**

**FileInputStream input = new FileInputStream(f);**

**FileInputStream** method **read** transfers of data from external storage to internal memory.

**InStream.read(text);**

The **FileWriter** class is used to transfer character-oriented data from internal memory to an external file. The **BufferedWriter** class manages line-oriented data. Together these two classes enable you to transfer line-oriented character strings.

**FileWriter outFile = new FileWriter("Java1706.dat");**

**BufferedWriter outStream = new BufferedWriter(outFile);**

These two statements can be "wrapped" together into a single statement with the same result using an anonymous object.\ for **FileWriter**.

**BufferedWriter outStream =**

**new BufferedWriter(new FileWriter("Java1707.dat"););**

**BufferedWriter** method **write** transfers character stream data from internal memory to external storage.

**outStream.write("Grace Hopper is my hero");**

**BufferedWriter** method **newLine** adds an **end-of-line** character to the external file. The result is that the next data will be stored on a new line in the text file.

**outStream.newLine();**

The **FileReader** class is used to transfer character-oriented data from an external file to internal memory. The **BufferedReader** class manages line-oriented data. Together these two classes enable you to transfer line-oriented character strings.

**FileReader inFile = new FileReader("Java1706.dat");**

**BufferedReader inStream = new BufferedReader(inFile);**

These two statements can be "wrapped" together into a single statement with the same result.

**BufferedReader inStream =**

**new BufferedReader(new FileReader(args[0]));**

**BufferedReader** method **readLine** transfers one complete line of characters, including the **end-of-line** character, from external storage to internal memory.

**inString = InStream.readLine();**

Static **String** method **valueOf** converts **intNumber** to a String object. This is not a file class method, but you need to use this method to store numerical values.

**String S = String.valueOf(IntNumber);**

Static **Integer** method **parseInt** converts **InString** to an integer value. This is not a file class method, but you need to use this method to convert numerical character values back to integers.

**rndInt = Integer.parseInt(inString);**