**ISTE-121 Day07 – BinaryStreamIO**

**Text vs. Binary IO**

Text files are sequences of characters. We can impose structure on them by defining a ‘record’ as one line of text (lines delimited by ‘\n’ characters) and we can separate the records into ‘fields’ by using a delimiter character of some sort (‘\t’, ‘:’, etc.). This division is sort of contrived, and it leaves the records and fields to be variable length. Still, people are most comfortable with text, as they naturally deal with text all the time.

**Text IO Refresher**

To read a text file, we usually use something like:

try {  
 // Construct a **File** object from the file name (a String)  
 // Then, open the File as a **FileReader**, and pass the  
 // **FileReader** to the **Scanner** constructor. Now, we can use  
 // convenient Scanner methods like nextInt(), nextLine(),   
 // etc.  
 Scanner scn = new Scanner(new FileReader(new File(fileName)));  
 while(scn.hasNextLine()) {  
 String line = scn.nextLine();  
 // Do something with the line  
 }  
 scn.close();  
 }  
 catch(IOException ioe) {  
 Alert alert = new Alert(AlertType.ERROR, "IO Exception " + ioe);

alert.setHeaderText(“IO Exception”);

alert.showAndWait();  
 "IO Exception", JOptionPane.ERROR\_MESSAGE);  
 return;  
 }  
  
and to write to a text file, we usually use something like this:

try {  
 // Construct a **File** object from the file name (a String)  
 // Then, open the File as a **FileWriter**, and pass the  
 // **FileWriter** to the **PrintWriter** constructor. Now, we can use  
 // convenient PrintWriter methods like print(), println(),   
 // printf(), etc.  
 PrintWriter pwt =   
 new PrintWriter(new FileWriter(new File(fileName)));  
 for(String s : arrayList) {  
 pwt.println(s);  
 }  
 pwt.close();  
 }  
 catch(IOException ioe) {  
 Alert alert = Alert(AlertType.Error, "IO Exception " + ioe);

alert.setHeaderText(“IO Exception”);

alert.showAndWait();  
 return;  
 }

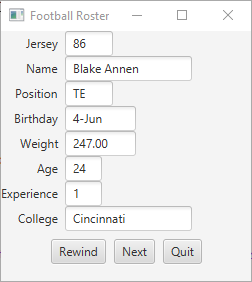
**Reading a Binary File**

In binary IO, we write Java native types directly to the file, without converting them to strings. This is usually faster and usually takes less space in the file. However, people cannot easily read binary data, so a program needs to be written to convert it to text. If the file is only for programs to use, however, binary makes good sense.

In today’s downloads, there is a file named FBRosterBIN.dat. This is a binary file with information about football players. It contains one record for each player. For each player, there are 8 ‘fields’:

|  |  |
| --- | --- |
| **Field Name** | **Data Type** |
| Jersey number | int |
| Name | UTF |
| Position | UTF |
| Birthday | UTF |
| Weight in pounds | double |
| Age in years | int |
| Experience | UTF |
| College | UTF |

We want to write a program that will read this file, record by record, and display it to the user. The GUI for this program is (see ReadFBRoster.java in today’s downloads):



When done, our program will open up the file FBRosterBIN.dat (the name is hard coded into the program) and display the first record. Each click of ‘Next’ will read in the next record and display it. On EOF, the program will announce EOF in an Alert, but will **not** change the GUI contents and will not quit. This will allow the user to click Quit or Rewind, if s/he likes.

The ‘Rewind’ button will close the file and reopen it to go back to the first player.

The ‘Quit’ button will terminate the program.

Make a copy of ReadFBRoster.java (from today’s downloads) called ReadFBRoster1.java (change the class name and constructor name and the call to the constructor in the main program).

**NOTES on the GUI**

There are a large number (8) of text fields in this program. Each needs to be displayed with a label. The label needs to be right justified. Each must have a different PrefColumnWidth. To simplify the code for this, we have placed all of the TextFields in an array of TextFields (tfAll). We have also placed the Strings for the labels in an array (tfLabels) and the widths of the text fields in an array (tfWidths). The TextFields and their placements are all handled in the following loop:

GridPane gpTop = new GridPane();  
 for(int i = 0; i < tfAll.length; i++) {  
 Label label = new Label(tfLabels[i]);  
 gpTop.setHalignment(label, HPos.RIGHT);  
 gpTop.setFillWidth(tfAll[i], false);  
 tfAll[i].setPrefColumnCount(tfWidths[i]);  
 tfAll[i].setEditable(false);  
 gpTop.addRow(i, label, tfAll[i]);  
 }  
 root.getChildren().add(gpTop);  
The setHalignment method of the GridPane tells the gridpane how the labels are to be aligned. The setFillWidth method, tells the GridPane **not** to expand the TextFields to the full size of the grid, but to adhere to the PrefColumnCount for the textfields.

**Opening the File**

Create attributes for the File IO we will be doing:  
 // File IO attributes  
 public static final String FILE\_NAME = "FBRosterBIN.dat";  
 private File fileObj = null;  
 private FileInputStream fis = null;  
 private DataInputStream dis = null;  
  
Similar to text io, we will create a **File** object, create a **FileInputStream** from that, and a **DataInputStream** from that:

try {  
 fileObj = new File(FILE\_NAME);  
 fis = new FileInputStream(fileObj);  
 dis = new DataInputStream(fis);  
 }  
 catch(IOException ioe) {  
 Alert alert = new Alert(AlertType.ERROR, "Cannot open file "   
 + FILE\_NAME + "\n" + ioe);  
 alert.showAndWait();  
 System.exit(1);  
 }  
  
The above code, of course, must be in a try catch, because things can go wrong with IO. Place the above code in the constructor, **before** setting the scene and showing the stage. Get this to compile, but don’t run it yet.

**Reading the Next Record (Player)**

Once open, the file can then be read. DataInputStream (our object ‘dis’) gives us a very useful variety of input methods:  
 readInt, readDouble, readChar, readBoolean, readUTF  
for reading binary data of different types. As long as we know the type of the next thing to be read, we can use the appropriate method to read it. These methods will read the proper amount of data and store it in memory as an int, double, char, boolean, or String. There are others, too: readLong, readFloat, readShort, for example.

Write a method, readNext(), to read the next record, and display its contents in the GUI. Call readNext() as the last line of the constructor and in handle in the case of the “Next” button. Run your program now. ‘Rewind’ and ‘Quit’ won’t work yet, but the rest should.

**Rewinding to the Start of the File**

The way to do this is to close the file and reopen it. To close it, simply:  
 dis.close();  
inside a try/catch. In the same try/catch, open it again (copy the code from above).

Run your program and see if this works for you.

You should be able to program the ‘Quit’ button which will complete this program.

**Writing a Binary File**

We can turn this around and write out a file as well. Look at the file WriteFBRoster.java in today’s downloads. Rename it WriteFBRoster1.java. It is the same GUI, except there are only two buttons now, ‘Write’ and ‘Close and Quit’. Also, the TextFields are now editable. The ‘Write’ button will write what’s in the TextFields to a file.

**Opening the File for Writing**

Create attributes for the File IO we will be doing. **NOTE**: use a different file name than in ReadFBRoster1, or you will overwrite the original file:  
 // File IO attributes  
 public static final String FILE\_NAME = "FBRosterBINOut.dat";  
 private File fileObj = null;  
 private FileOutputStream fos = null;  
 private DataOutputStream dos = null;  
  
Similar to the above, we create a **FileOutputStream** from the File object, and pass that to **DataOutputStream** constructor:

try {  
 fileObj = new File(FILE\_NAME);  
 fos = new FileOutputStream(fileObj);  
 dos = new DataOutputStream(fos);  
 }  
 catch(IOException ioe) {  
 Alert alert = new Alert(AlertType.ERROR, "Cannot open file "   
 + FILE\_NAME + "\n" + ioe);  
 alert.showAndWait();  
 System.exit(1);  
 }  
The above code, of course, must be in a try catch, because things can go wrong with IO. Place the above code in the constructor, before setting the scene and starting the program. Get this to compile, but don’t run it yet.

**Writing the Next Record (Player)**

Once opened, the file can then be written. DataOutputStream (our object ‘dos’) has methods similar to those from DataInputStream:  
 writeInt, writeDouble, writeChar, writeBoolean, writeUTF (and others)  
for writing binary data of different types. These methods will write the proper amount of data from memory to the file, in binary.

Write a method, writeRecord(), to write one record to the file. It will write the fields in the order shown, and use the proper ‘write’ method for the type. Strings are written with writeUTF.

**NOTE:** all TextFields contain Strings (returned by the getText() method). You must convert these to the proper type in the case of non-String fields before writing. Use Integer.parseInt() and Double.parseDouble() to do this.

Also, after writing the record out, your program should clear all of the TextFields. Unlike FBRoster1.java, **do NOT** call writeRecord() as the last line of the constructor. However, **DO** call it in handle in the case of the “Write” button. Run your program now.

Finally, program the Close and Quit button. To close the file (necessary if the data is to be finalized) use the code:

try {  
 dos.close();  
 }  
 catch(IOException ioe) {  
 Alert alert = new Alert(AlertType.ERROR,   
 "Cannot close file " + FILE\_NAME + "\n" + ioe);  
 alert.showAndWait();  
 }  
 System.exit(1);  
Try your program out. Run it, make up data, and click ‘Write’ to put 3-4 players in the file. When done, click ‘Close and Quit’ to finalize things. If you now rename FBRosterBINOut.data to FBRosterBIN.dat, you can run ReadFBRoster1.java to see if the file was created correctly.

**Writing Issues**

When you open a file **for reading** that does not exist, an exception is thrown. Try this: delete or rename BBRosterBIN.dat and run ReadFBRoster1.java. What happens?

When you open a file **for writing** that does not exist, no error occurs. Instead, the file is created (it is empty) and the program continues (of course there are other errors that might occur, such as the device is full or the folder you are working in does not have permission for you to create a new file).

If you open a file for writing and it **does** exist, then it is overwritten (replaced). This means the file is emptied and you write new data in place of the old data.

Another option would be to **append** to the file. This can be accomplished, if you like. To do this, at the step where you open the FileOutputStream, you must provide a second parameter (after the File object) that is a boolean. Use **false** to overwrite (replace) the file, or **true** to append to it. Appending means that existing data is preserved, and new data is written at the end of the old data.