Week 9 - Friday

COMP 2000

Last time

- What did we talk about last time?
- Reading and writing text files
- Error handling

Questions?

Project 3

More file practice

- Prompt the user for an input file name
- If the file isn't accessible, prompt the user to enter the name again
- Read in all the integers in this file (until you run out)
- Close the file
- Store all the values in an **`ArrayList<Integer>`**
- Sort them
- Find the mode (the value that appears the most)

Binary Files

What is a binary file?

- Technically, **all** files are binary files
  - They all carry data stored in binary
- But some of those binary files are called **text files** because they are filled with human readable text
- When most people talk about binary files, they mean files with data that is only computer readable

Why use binary files?

- Wouldn't it be easier to use all human readable files?
- Binary files can be more efficient
  - In binary, all **`int`** values are 4 bytes
- In text, they can take up a lot more
- In text, you also need a space or other separator to divide the numbers

Integer	Bytes in text representation
0	1
92	2
789	3
4551	4
10890999	8
204471262	9
-2000000000	11

Most files are binary files

- Because they have a representation that is more compact (and more similar to how data is stored in your program), most files are binary (non-human-readable) files
- Many media files start with **metadata**
  - Format information
  - Size
- Then, they have the actual data (RGB values, audio samples, frames of video, etc.)
- Binary files include most common file formats: **`.jpg`**, **`.png`**, **`.mp3`**, **`.avi`**, **`.pdf`**, **`.docx`**, **`.pptx`**, and on and on

Reading binary files

- Reading from binary files uses a completely different set of objects than reading from text files
- We create a **`DataInputStream`** from a **`FileInputStream`**
- The **`FileInputStream`** takes the name of the file path

```
DataInputStream in = new DataInputStream(new
  FileInputStream("input.dat"));
```

- You can create a **`FileInputStream`** first on a separate line, but there's no need to do so

Reading data

- Typically, we will read in individual pieces of data in binary from a **`DataInputStream`**

Return Type	Method	Use
`boolean`	`readBoolean()`	Read a single **`boolean`**
`byte`	`readByte()`	Read a single **`byte`**
`char`	`readChar()`	Read a single **`char`**
`double`	`readDouble()`	Read a single **`double`**
`float`	`readFloat()`	Read a single **`float`**
`int`	`readInt()`	Read a single **`int`**
`long`	`readLong()`	Read a single **`long`**
`short`	`readShort()`	Read a single **`short`**
`int`	`read(byte[] array)`	Read **`byte`** values into **`array`**, return the number read
`int`	`skipBytes(int n)`	Skip at most **`n`** bytes in the stream, return the number skipped

Example summing `double` values

- The following code assumes that a file contains starts with an **`int`** value giving the number of **`double`** values that come after it

```
DataInputStream in = null;
try {
   in = new DataInputStream(new FileInputStream("numbers.dat"));
   int length = in.readInt();
   double sum = 0.0;
   for(int i = 0; i < length; ++i)
       sum += in.readDouble();
   System.out.println("Sum: " + sum);
}
catch(IOException e) {
   System.out.println("File problems!");
}
finally {
   try{ in.close(); } catch(Exception e){}
}
```

Error handling

- The reading methods in **`DataInputStream`** can throw:
  - **`EOFException`** if the end of the file was reached but you still try to read something
  - **`IOException`** if the stream was closed (or something else goes wrong)
- Since **`EOFException`** and even **`FileNotFoundException`** are both children of **`IOException`**, it's possible (as we did on the previous slide) to have a single catch block that handles an **`IOException`**

Closing the file

- As with text files, we closed our files in a **`finally`** block
- You might have noticed that there was a baby **`try-catch`** block inside of there as well

```
finally {
   try{ in.close(); } catch(Exception e){}
}
```

- For whatever reason, closing a **`DataInputStream`** can throw an **`IOException`**
- By having a **`try-catch`** that will catch anything, we deal with the **`IOException`** as well as catching the **`NullPointerException`** that happens if we try to close a **`null DataInputStream`**
- Is that a good idea?
- Eh…it's fine: We're just trying to close the file and not crash our program

Writing binary files

- Writing to binary files is very similar to reading from binary files
- We create a **`DataOutputStream`** from a **`FileOutputStream`**
- The **`FileOutputStream`** takes the name of the file path

```
DataOutputStream out = new DataOutputStream(new
  FileOutputStream("output.dat"));
```

- The writing methods are similar too

Writing data

- Typically, we will write out individual pieces of data in binary with a **`DataOutputStream`**

ReturnType	Method	Use
`void`	`writeBoolean(boolean value)`	Write a single **`boolean`**
`void`	`writeByte(byte value)`	Write a single **`byte`**
`void`	`writeChar(int value)`	Write a single **`char`**
`void`	`writeDouble(double value)`	Write a single **`double`**
`void`	`writeFloat(float value)`	Write a single **`float`**
`void`	`writeInt(int value)`	Write a single **`int`**
`void`	`writeLong(long value)`	Write a single **`long`**
`void`	`writeShort(int value)`	Write a single **`short`**
`void`	`write(byte[] values)`	Write all the **`byte`** values from **`values`**

Example writing double values

- The following code assumes that a file starts with an **int** value giving the number of **double** values that come after it

```
DataOutputStream out = null;
try {
   out = new DataOutputStream(new FileOutputStream("numbers.dat"));
   out.writeInt(100);
   for(int i = 0; i < 100; ++i)
      out.writeDouble(Math.random() * 1000);
}
catch(IOException e) {
   System.out.println("File problems!");
}
finally {
   try{ out.close(); } catch(Exception e){}
}
```

Putting the I/O together

- File input and output need to match each other well, especially for binary I/O
- If data values are out of order, you'll get garbage, and it'll be hard to know why
- Once you write the file output code, you can easily copy and paste it to write the input code
  - Change every **`out`** to **`in`**
  - Change every **`write`** to **`read`** (and move the method arguments to save return values)
- The structures are parallel

Comparison of binary and text files

- Write a program that:
  - Prompts the user for a file name
  - Opens the file as a text file
  - Writes the first 1,000 perfect cubes: 1, 8, 27, 64, etc. as text
  - Closes the file
- Write a second, similar program that:
  - Prompts the user for a file name
  - Opens the file as a binary file
  - Writes the first 1,000 perfect cubes: 1, 8, 27, 64, etc. in binary
  - Closes the file
- What do the files look like inside?
- How do the sizes of the files compare?

Reading and Writing Whole Objects

What if I wanted to read or write a whole object?

- An object has data inside of it
- Each piece of data is either a reference to an object or is primitive data
- When reading or writing whole objects, we could read or write each piece of data separately
- But doing so is challenging because we could forget some data
- And because there could be circular references:
  - Object A might have a reference to object B which might have a reference to object A again…

Serialization

- If only there was some magical way to read or write a whole object at once…
- There is!
- It's called **serialization**
- Serialization takes a reference to an object and dumps it into a file
- It writes representations to primitive types pretty much the same way that a **`DataOutputStream`** does
- And if there're objects inside of the object you're serializing, it serializes them too
- **And!** Serialization makes a note of all the objects that are getting serialized, so if it sees an object a second time, it just writes down a serial number for it instead of the whole thing

`Serializable` interface

- Serialization is one of the closest things to magic you'll see in programming
- You only need to implement the **`Serializable`** interface on your object
  - And the **`Serializable`** interface has no methods!
- It's just a way of marking an object as reasonable to try to dump into a file
- Most objects are reasonable to dump into a file!

Example Serializable class

- Here's a class we might want to be able to dump into a file

```
public class Troll implements Serializable {
  private String name;
  private int age;
  private Object hatedThing; // All trolls hate something
  public Troll(String name, int age, Object hatedThing) {
     this.name = name;
     this.age = age;
     this.hatedThing = hatedThing;
  }
  public Object getHatedThing() {
     return hatedThing;
  }
}
```

Writing using serialization

- To write an object marked **`Serializable`**, you need to create an **`ObjectOutputStream`**
- You create an **`ObjectOutputStream`** the same way that you create a **`DataOutputStream`**, by passing in a **`FileOutputStream`**
  - At this point, you might be wondering why all these objects take **`FileOutputStream`** objects and can't take just take a **`File`** object or even a file name
  - In actuality, you can pass in any **`OutputStream`** object (of which **`FileOutputStream`** is a child), like maybe one that sends the data across the network instead of storing it into a file
- An **`ObjectOutputStream`** object has many methods, but the only one that matters is **`writeObject()`**
- Pass your object to that method and it'll get written out in its totality, no fuss

Example of writing

- Here's some code that creates a couple of **`Troll`** objects and then writes them to a file called **`trolls.dat`**

```
Troll tom = new Troll("Tom", 351, "Bilbo Baggins");
Troll bert = new Troll("Bert", 417, tom);
ObjectOutputStream out = null;
try {
    out = new ObjectOutputStream(new FileOutputStream("trolls.dat"));
    out.writeObject(tom);
    out.writeObject(bert);
}
catch(IOException e) {
    System.out.println("Serialization failed.");
}
finally { try{ out.close(); } catch(Exception e){} }
```

Reading using serialization

- To read objects that have been serialized to a file, you need to create an **`ObjectInputStream`**
- You create an **`ObjectOutputStream`** the same way that you create a **`DataInputStream`**, by passing in a **`FileInputStream`**
- For each object serialized, you call the **`readObject()`** method to restore it from the file
- Note that **`readObject()`** has a return type of **`Object`**, so you'll need to cast your object if you want to store it in a reference of its own type

Example of reading

- Here's some code that reads in the **`Troll`** objects we serialized in the previous example

```
Troll tom = null;
Troll bert = null;
ObjectInputStream in = null;
try {
    in = new ObjectInputStream(new FileInputStream("trolls.dat"));
    tom = (Troll)in.readObject();
    bert = (Troll)in.readObject();
}
catch(IOException e) {
    System.out.println("Deserialization failed.");
}
finally { try{ in.close(); } catch(Exception e){} }
```

The good

- Serialization allows you to read or write objects (even complex objects) or arrays of objects in a single line of code
- It's an impressive achievement of Java
- To make your own classes serializable, all you have to do is mark them with the **`Serializable`** interface
  - An interface with no methods!
- It more or less works like magic!

The bad

- Some objects are not serializable, but they are comparatively rare
- An example is the **`Thread`** class, which encapsulates the state of a currently running thread…so how could you store it on disk?
- Serialization does have storage overhead needed to keep track of the size of arrays and type information about classes
  - You might be able to use less space if you stored the data directly

The ugly

- If you forget to mark one of your classes **`Serializable`**, it will crash your code when you try to write it out, even indirectly
- If you serialize objects to a file but later change the class, adding or removing members or methods, you will no longer be able to read those objects back from the file
- Their data in the file will no longer match what the class is supposed to look like
- This problem can happen with different versions of the same program

Upcoming

Next time…

- Networking basics:
  - IP addresses
  - Ports
  - Sockets

Reminders

- **Work on Project 3**
  - **Form teams if you haven't!**
  - **Project 3 is now due on April 3**
- Read Chapter 21