COMP1006/1406 - Summer 2016 - Tutorial 4

Objectives: Basic File I/O.

Exception Handling

File I/O

There are several ways in which Java allows you read and write to files. There are classes for reading/writing binary files and there are classes for reading/writing text files.

Whether you are reading from or writing to a file (or both), the following pattern should always be followed:

- open the file
- read from or write to the file
- close the file

It is important that you always close a file when you are done reading it or writing to it. Can you think of any reasons why this is important?

If you want some extra resources for file i/o in Java, see Chapter 11 of Dr. Lanthier's notes. It contains notes and examples for writing/reading binary data, text data, objects and working with `File` objects.

`http://people.scs.carleton.ca/~lanthier/teaching/COMP1406/Notes/COMP1406_Ch11_FileIO.pdf`

1: Reading Text Files

There are many ways to read data from files in Java. Two basic ways of reading text data is to the use the `Scanner` class and the `BufferedReader` class.

Regardless of which class you use to read the data, you will need to tell that class which file you want to read from. We'll use the `FileReader` class for this. It has a simple constructor that accepts a String which is the file's name.

```
String fileName = "example.txt";
try{
  FileReader file = new FileReader( fileName );
}
catch(FileNotFoundException e){
  System.err.println("Error: could not find file \"" + fileName + "\"");
}
```

We wrap the code in a `try`/`catch` beccause the constructor for the FileReader class potentially throws the `FileNotFoundException` exception.

```
public FileReader(String fileName) throws FileNotFoundException
```

(see `http://docs.oracle.com/javase/8/docs/api/java/io/FileReader.html`)

`Scanner`: Using a scanner object to read data from a file is essentially the same as using a scanner object to read data from standard input. The only differences are that

- We need to tell the scanner object "where" to read the data from. Instead of passing `System.in` (the standard input) into the constructor, we specify a file.
- We'll have to deal with exceptions and exception handling.

Using scanner for file input is good when you want to read in single words (strings) and numbers from the input file. We just have to be careful because Scanner will throw an exception if you try to read data after you have already reached the end of file (EOF). You can use the different hasNext methods to check if you have more data to read.

`BufferedReader`: A bufferedreader object will read a text file character by character (returning the unicode value of the character) or will read a file line by line. It reads lines of the file as Strings.

When you try to read a line (`readLine()`) beyond the end of a file, the method will return `null`. You can just read a file line by line until you get a null back from the method.

➡ Download the `Read.java` program; compile and try to run it.

➡ Modify the `Read.java` program so that it reads all the integers from the file `"data.txt"`, and outputs, to standard output, the average value of all the numbers in the file. Your program should use a `Scanner` object to read all the data.

➡ Modify the `Read.java` program so that it uses a `BufferedReader` object to read all the data from the files (and still outputs the average of all the numbers).

2: Writing Text Data

The `Write` class in Java has several subclasses to let you write data to text files. The `BufferedWriter` class is used for writing lines of text, the `FileWrite` is used for writing characters, and the `PrintWriter` class lets us write strings (just like the `PrintStream` class, the `out` in `System.out`, lets us print to the screen).

Regardless of which class you use to write the data, you will need to tell that class which file you want to write to. We'll use the `FileWriter` class for this. It has a simple constructor that accepts a String which is the file's name.

```
String fileName = "example.txt";
try{
  FileWriter file = new FileWriter( fileName );
  ...
}catch(FileNotFoundException e){
  System.err.println("Error: could not find file \"" + fileName + "\"");
}catch (IOException e) {
  System.out.println("Error: Cannot read from file");
}
```

We will consider only the `PrintWriter` class for outputing to a text file. Look at the API for the methods that it provides to us.

`http://docs.oracle.com/javase/8/docs/api/java/io/PrintWriter.html`

You should notice that it has the same methods that we used for printing to standard output: `print` and `println`. Further, we can pass any object to these methods and it will print the results of calling `toString()` on the objects to the file.

➡ Modify your `Read.java` program from above to do the following: read all the integers from the file data.txt file, sort the numbers, write the sorted numbers to the file `"sortedData.txt"`.

➡ Modify your `Read.java` program from above to do the following: read all the integers from the file data.txt file, sort the numbers, write the sorted numbers back to the file `"data.txt"`.

That is, your program should read all the data from a given file and then write the data back (in sorted order) to a file with the same name.

3: Objects I

Since most Java programs use objects to store data, we should be able to read/write object data using files. We'll consider a very simple way of doing this here. (We'll look at a better way in the next section.)

Consider the `Bunny` class provided. The **state** of a bunny object is determined by its two attributes: its name and age. This is all the information that we would need to store in a file to record the state of a given bunny.

We can store bunny information in a text file that looks like

```
Sam  12
Luis  13
Jose 7
```

Each line consists a name followed by some (non-newline) whitespace followed by an age. Alternatively, we could put each piece of data on its own line.

```
Sam
12
Luis
13
Jose
7
```

In either case, we are storing the data for three bunny objects.

Reading: To read a bunny object from the file, we need to read both the name and age, then call the Bunny constructor with this data to re-create the bunny. For this to work, we have to know the exact structure of the file.

Writing: To save a bunny object to the file, we need to simply write the name and age. In order for this file to be useful (so we can read the bunny data later), we must write the data in some specified structure. (For example, name always comes before age.)

When reading/writing object data with a file in this way, we need to be careful that we know the exact structure of the files that will store the data.

➡ Write a small program that reads in a collection of bunny objects from a file. It should then sort them and save the data back to another file.

4: Objects II

Reading and wiring objects using the simple method from the last section has some potential problems with it. Can you think of what these problems might be?

Let's look at a (potentially) better way of reading/writing objects using a file. The `ObjectOutputStream` and the `ObjectInputStream` allows us to read and write objects directly (without having to explicitly read/write all attribute data). In order for these classes to work though, the object you want to read/write must implement the `Serializable` interface.

The `Serializable` interface is an interface that has no attributes or methods declared. It is simply used as a flag to keep track of which classes are serializable and which are not. When a class is flagged as serializable, there will be a default ordering to its data when it is written or read. We won't have to deal with this. The `ObjectOutputStream` and the `ObjectInputStream` will use this default ordering for us.

➡ Download the `BunnyWrite.java` program. Try to compile and run this. Fix the bug in the code so that it works. The program should create a single bunny object and save it to a file (Bunny.dat).

When you get this working, open the Bunny.dat file in a text editor. You'll notice that the file created is not a plain text file.

➡ Download the BunnyRead.java program and see that it reads the bunny object stored from the BunnyWrite program. You'll first need to fix the bug in the code.

➡ Modify the Bunny class so that each Bunny has the following additional attributes: an arraylist of bunny objects called children (these will be all children of this bunny), and a Home object called home (which is where this bunny lives). Create a simple House class that stores a string for the address of the house.

Check that your BunnyRead and BunnyWrite programs work with your new Bunny class.

Extra Reading 

The ObjectOutputStream and the ObjectInputStream are a great way to read/write object data using files in a consistent way. One downside to this is that the files that store the object data are generally not human-readable. We can read parts of it, but it is a format that was not designed to be read by humans.

It would be nice to be able to store our objects in a format that could be easily read by humans. For this to be useful, there must be a standard for storing the object data. One such standard is the the JSON (Javascript Object Notation) format (`http://json.org/`). This is a very common format of sending object information in Internet applications.

➡ Using json-simple class modify the bunny programs to save some bunny objects in the json format and then read them back from the text file.

You will need to download the json-simple .jar file and put this in your directory. (`http://json-simple.googlecode.com/files/json-simple-1.1.1.jar`) Once you have the jar file, extract its components with

```
jar xf json-simple-1.1.1.jar
```

The following webpage has simple example of writing and reading json objects with json-simple. (`https://www.mkyong.com/java/json-simple-example-read-and-write-json/`)