File I/O in Java

CSC110 / CSC205

Adapted from notes by Pat Baker based on Java Foundations by Lewis, Chase, & DePasquale

Topics

- Reading data from a file
- Writing data to a file
- Parsing data

Reading from a File

Using Files for Data Storage

- We can use a **file** rather than reading console input entered from the keyboard
- Files are stored on **secondary storage** such as your computer's hard drive or a flash drive
- Files allow data to be retained before and after program execution

File Types and File Access

There are two types of files to consider:

1. **Streams** or **Sequential Access Files**: Data is accessed in order or sequentially

2. **Random Access Files**: Code can read/access a particular position in a file

We will be working with streams. We will not cover random access files.

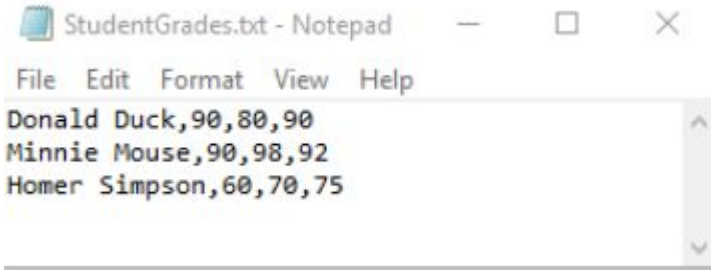
Streams: Sequential Access Files

When data is read sequentially, it means that the code reads the 1st piece of data, then reads the 2nd piece of data, then reads the 3rd piece of data ... then reads the last piece of data. So, to access the last piece of data, you have to read all of the previous data.

- Java views your file as a series of bytes - Picture bytes flowing into your program from an input device through a stream, which acts like a pipeline
- In other words, data arrives into your application via an **input stream**
- In Java, a stream is an object and like all objects, streams have data and methods
- `System.in` is an input stream

Streams & Text Files

Text files - what are they? Text files contain letters, digits and punctuation. It can be viewed with a text editor such as Notepad. They can also be viewed in Eclipse.
In our examples, we will use an input stream object to read from a text file.
Example: StudentGrades.txt is a file that contains the name and 3 exam scores of some terrific CSC110 students



Reading Data In From a File

Summary

1. Make necessary Java classes available
2. Open the file and associate it with an object
3. Read data in from the file until the end of the file is reached
4. Close the file

We will go into more detail in the next few slides

Reading Data In From a File

1. Make necessary Java classes available

```
import java.io.*    //needed for FileReader
import java.util.Scanner  //needed for Scanner
```

2. Open the file and associate it with an object. We will be reading in data from this file
   Create an input file object, associate the object with a physical file and open the file

```
Scanner inFile = new Scanner(new FileReader("src/ch10Files/StudentGrades.txt"));
```

   This associates the object inFile with the physical file StudentGrades.txt and opens it.

   - The filename may include drive, path info
   - The input file *must* exist or an exception will be thrown
   - Note: **`FileReader`** objects are input streams. We use **`Scanner`** to assist in processing the stream

Reading Data In From a File

3. Read data in from the file until the end of the file is reached.
    - Using a **`while`** loop, read in a line of data.
    - Only enter the loop if there is more data to be read.
    - Since you are using a **`Scanner`** object, your code can use the **`Scanner`** methods to
        - Check if there is more data to be read: **`hasNextLine()`**
        - Read a line of data: **`nextLine()`**

```
while ( inFile.hasNextLine() ) {
    line = inFile.nextLine();
}
```

Reading Data In From a File

Full code example:

```
String line;   // variable to hold each line of text

// will return true and enter the loop if there is another line in the file
while ( inFile.hasNextLine() ) {
    line = inFile.nextLine();  //read a line of data from the file
                                  and put it in the variable line
    //any additional processing of each line read
}
```

Reading Data In From a File

4. Close the file.

   Close the Scanner object inFile and therefore the associated physical file.

```
inFile.close();
```

Writing Out to a File

Using Files for Data Storage

- We can use a file to store output rather than displaying output to the console
- Files are stored on secondary storage such as your computer's hard drive or a flash drive
- Files allow data to be retained before and after program execution

Streams: Sequential Access Files

Our output files will also be Streams, which means your code will write to the file sequentially. Your application will write the 1st piece of data, then write the 2nd piece of data, write the 3rd piece of data, ... then write the last piece of data.

- Recall that Java views your files as a series of bytes. Picture bytes flowing to an output device through a **stream**, which acts like a pipeline.
- In other words, data will be written to a file via an **output stream**.
- In Java, output streams are objects with data and methods.
- **`System.out`** is an output stream associated with your console. It has methods **`print`** and **`println`**.

```
System.out.println("Hi Class!");
```

Streams & Text Files

In our examples, we will use an output stream object to write to a text file.

Recall that text files contain letters, digits and punctuation. They can be viewed with a text editor such as Notepad. They can also be viewed in Eclipse.

Example: GradeReport.txt is a file that contains the output that your code generates. Your code will write out each line in order.

Writing Data Out To a File

Summary

1. Make necessary Java classes available
2. Open the file we are writing data out to
3. Write the data out to the file
4. Close the file

We will go into more detail in the next few slides

Writing Data Out To a File

1. Make necessary Java classes available.

```
import java.io.*   //needed for PrintWriter, which is an output stream
```

2. Open the file. We will be writing data out to this file.

   Create an output file object, associate the object with a physical file and open the file.

```
PrintWriter outFile = new PrintWriter("src/ch10Files/GradeReport.txt");
```

   This associates the object outFile with the physical file GradeReport.txt and opens it for writing. (if the file already exists, will write over the file)

   The filename may include drive, path info

Writing Data Out To a File

3. 3. Write data out to the file.

   This example writes a String called line to the file.

   ```
   outFile.println(line);
   ```

   Notice how the PrintWriter object outFile behaves like System.out.

4. Close the file.

   Close the PrintWriter object outFile and therefore the associated physical file.

   ```
   outFile.close();
   ```

Parsing Data

Parsing / Tokenizing

- Sometimes your code needs to take a line of data and break it apart
- This is called **parsing** or **tokenizing**
- The `Scanner` class is very useful to break apart or parse a line of data
- Those separate items are also called **tokens**
- Before your code can parse a line of data, you will need to know ahead of time how the file is organized
- Specifically, you will be interested in how the data is separated or **delimited**

Breaking Apart a String Using `Scanner`

- A common way data in a file is broken up (**tokenized**) is by using a comma to separate the data
  - In this case we call the comma the **delimiter** or separator
- The data for our activity looks like the following:

```
Donald Duck,90,80,90
Minnie Mouse,90,98,92
Homer Simpson,60,70,75
```

- Each “chunk” of data is separated by a comma
- Each line has a student name and 3 exam scores
- Notice all lines have the same structure – *this is very important*

Breaking Apart a String Using `Scanner`

If the comma is our delimiter, we have 4 pieces of data in our file - A name and 3 exam scores (`Donald Duck,90,80,90`)

```
String line;    // your code already has this
String name;    // used to hold the student's name
int exam1, exam2, exam3;  //to hold the student's exam scores
while ( inFile.hasNextLine( ) ) //will return true if there is another line in the file
{
            line = inFile.nextLine();              //read a line of data
            Scanner tokens = new Scanner(line);    //create another Scanner called tokens associated with line
            tokens.useDelimiter(",");              //use a comma as the delimiter

            name = tokens.next();                  //read in the first chunk of data until a comma is encountered
            exam1 = tokens.nextInt();              //read in the second chunk of data until a commas is encountered
            exam2 = tokens.nextInt();              //and so forth….you get the idea
            exam3 = tokens.nextInt();

      //At this point all the data is separated into variables & we can use the variables as needed
      System.out.println("Student: " + name + "    Java Exam 1: " + exam1);
}
```

Breaking Apart a String Using `String.split()`

Another way to break apart a `String` in Java is to use `String.split()`

```
String phrase = "Grace Hopper wrote the first compiler";

// create an array of Strings called words.
//   each element in the array will contain one of the words in phrase
String[] words = phrase.split(" ");  //set the space as the delimiter

System.out.println("The phrase has been parsed into " + words.length + " words.");
System.out.println(words[0]);
System.out.println(words[1]);
System.out.println(words[2]);
System.out.println(words[3]);
System.out.println(words[4]);
System.out.println(words[5]);
```


```
The phrase has been parsed into 6 words.
Grace
Hopper
wrote
the
first
compiler
```

Parsing comparison between `split()` and `Scanner`

- **`String.split( )`** breaks a **`String`** into smaller **`String`**s

- Associating a **`String`** made up of multiple words with **`Scanner`**, provides a way to read in integers, doubles and other data types

Now go write some code!