COP 3330 – Object Oriented Programming in Java
Reading Input From a File

Use of Scanner for Files

A Scanner object can read input from various sources. So far, we've only used a Scanner to read in input from the keyboard, but by passing the constructor a File, we can instead set it up to read from a file. Assuming that the file is in the same directory as the .java file, here is how we can set up a Scanner object to read from a file, whose name is stored in the String filename:

```
Scanner fin = new Scanner(new File(filename));
```

This creates a Scanner object with the reference name fin, that will start reading from the file with the name filename (assuming this is a String variable storing the name of the file to read from.) Note: To use a file, you must import that java.io.* package, since File is in the IO package.

When we read from a file, we must report that we may throw and IOException. (This isn't necessary when we read from standard input.)

Since we've already seen what calls are in the Scanner class, there's basically nothing new to learn. Either read through the file line by line, calling the nextLine() method, OR read through the file token by token using nextInt(), nextDouble() and next().

If the file format doesn't clearly indicate the number of lines or tokens, then the Scanner method:

```
// Returns true if this Scanner has another token left in its
// stream, and false otherwise.
hasNext()
```

is very useful.

Another difference between reading from standard input and a File is that we must close the Scanner object that reads from a file after we are done reading from the file with the close() method.

We'll look at three examples in this lecture:

(1) Read in a dictionary file and then ask the user to enter words to search for. We'll use Java's built in binary search method to determine if the string entered by the user is a real word (acccording to our dictionary).

(2) We'll rewrite our solution to the problem, "Sorting Student Presentations" so that it reads input from a file instead of from standard input.

(3) We'll look at an example where we read input from a large weather file and process some statistics about that data.

Dictionary and Binary Search

In this example, we'll ask the user to enter the file name for a dictionary. We will assume that the first line of this file will have a single positive integer, n, representing the number of words in the dictionary, and then each word follows, one per line, with lowercase letters only. We'll also assume that these words are already in alphabetical order. (Though, if they weren't, all we would have to do is call either Arrays.sort.)

After we load the dictionary into an array, we can ask the user to enter a word. We can then use the built in binarySearch method in the Arrays class to look for the word the user entered in the dictionary.

We'll need the following two imports:

```
import java.util.*;
import java.io.*;
```

To allow the user some grace in entering the file name, we'll also try to catch a FileNotFoundException. Since we try to catch this exception, we don't need to throw an IOException. If we didn't try to catch this exception, we would have to report from main that we might throw an IOException.

Here's the beginning of the main method:

```
public static void main(String[] args) {

    Scanner stdin = new Scanner(System.in);
    System.out.println("Please enter your dictionary file.");
    String filename = stdin.next();
    Scanner fin = null;

    while (true) {

        try {
            fin = new Scanner(new File(filename));
        }
        catch (FileNotFoundException e) {
            System.out.println("Sorry, that file wasn't found. Try again.");
            filename = stdin.next();
        }

        if (fin != null) break;
    }
```

As you can see, after reading in the dictionary file initially, we do a loop with a try-catch in it, just in case the file name isn't good. In the catch, we'll get the next file name the user enters, if the previous one was invalid.

Loading the dictionary is pretty quick. We just have to read in the number of words, then use that to allocate the array, then read in each word into the array. After we're done, we can close the file. Here is the corresponding code:

```
int numWords = fin.nextInt();
String[] words = new String[numWords];
for (int i=0; i<numWords; i++)
    words[i] = fin.next();

fin.close();
```

Finally, in the last part of the code we let the user enter words to search for, until they don't want to. Here we use the built in binarySearch method in the Arrays class. This method takes in the array to search in, followed by the object to search for. If it returns a non-negative index, then the item being searched for is in the array at the index indicated. So, for our purposes, we just check if the value returned by binarySearch is greater than or equal to 0. Here is code that accomplishes this:

```
System.out.println("Do you want to enter a word to search for?");
String response = stdin.next().toLowerCase();

while (response.charAt(0) == 'y') {

    System.out.println("What word do you want to search for?");
    String word = stdin.next();

    int idx = Arrays.binarySearch(words, word);

    if (idx >= 0)
        System.out.println("Yes, that's a valid word.");
    else
        System.out.println("Sorry, that's not a valid word.");

    System.out.println("Do you want to enter another word?");
    response = stdin.next().toLowerCase();
}
```

We process as many requests as the user wants, ending when they don't enter a string that starts with y. In the loop, we read in the word the user wants to search for and then call the binarySearch method, using its return value to determine what to print out for the user. In short, when the search value is in the array, the binarySearch method returns a non-negative integer representing the index where the search value was found. Thus, four our purposes, whenever this return value is non-negative, we can report that the word searched for was found.

Sorting Student Presentations

This problem is from an old UCF High School programming contest. Although the contest now uses standard input for its format, back in 2008 when this question was created, the protocol was to read the input from a file and output the results to standard output.

A summary of the question is as follows:

Instead of sorting names (single names all uppercase letters) by alphabetical order, sort them as follows:

If name N_1 has more A's than name N_2 , then N_1 comes before N_2 for this sorted order.
If there is a tie, we move onto comparing B's, then C's, ...
The input is guaranteed to have no anagrams (rearrangements of the same letters) so that these rules will be sufficient to break all ties when comparing any pairs of names.

Thus, this question is precisely a custom sorting question.

To solve this question using Java's built in sort, we need to create a student object that implements Comparable<student>. This object needs to have easy access to frequency information: the number of times each letter appears in a name. To that end, one of the instance variables of the object will be an array of size 26 names freq, where freq[i] stores the number of occurrences of letter i. (When i=0 the letter is 'A', when i=1, the letter is 'B', and so forth.)

Once we design the student class appropriately, solving the problem becomes fairly straightforward. The key is both the constructor and the compareTo method. In the constructor, after copying the name into the appropriate instance variable, we need to then loop through the letters of the name and set up the frequency array. Given that the string storing the name is s, then we can access the appropriate index in the frequency array when assessing the character in index i as follows:

```
freq[s.charAt(i)-'A']++;
```

The charAt method call accessess the appropriate character, then to convert it to the range 0 – 25, we subtract out the Ascii value of 'A'. This expression is the desired index to the frequency array. We then increment this int variable. Here is the beginning of the class with the instance variables and constructor:

```
class student implements Comparable<student> {

      private String name;
      private int[] freq;

      public student(String s) {
            name = s;
            freq = new int[26];
            for (int i=0; i<s.length(); i++)
                  freq[s.charAt(i)-'A']++;
      }
```

Now, let's get to the compareTo method. This only needs to access the frequency array of both this object and other (parameter passed in). We want to look at the difference at each individual index between the two arrays. Any time that difference isn't 0, we should return our result.

For example, if this.freq[0] = 3 and other.freq[0] = 2, we want this to come first, meaning that we need to return a negative value. Thus, it follows that when the two corresponding values in the freq array aren't equal, we need to return:

```
other.freq[i] - this.freq[i]
```

Here is the code for the compareTo method (in the version posted online, the keyword this is omitted):

```
public int compareTo(student other) {

    for (int i=0; i<freq.length; i++)
        if (freq[i] != other.freq[i])
            return other.freq[i] - freq[i];

    return 0;
}
```

This class also has a toString() method for convenience which just returns the name.

Now, we can look at the main method in the sorting class. This is where all the I/O happens, specifically where we read from the input file sorting.in. In this code, since we DON'T try to catch the IOException, we must simply report that main could throw it. Here is the set up of main and all of the changes due to the file I/O:

```
public class sorting {

    public static void main(String[] args) throws IOException {

        Scanner fin = new Scanner(new File("sorting.in"));
        int numCases = fin.nextInt();

        // Code to solve problem.

        fin.close();
    }
}
```

So, the differences are:

1) Reporting that main throws an IOException.
2) Passing the Scanner constructor a File object.
3) Closing the Scanner object.

Now, let's move onto the code to solve the problem (which uses the student class we just wrote).

We'll have a loop to process the cases. Inside this loop we read in the number of names. Then we'll create a student array of the appropriate size, and set up a loop to read in the names. As we read in each name, we call the student constructor to create that student object. Once that loop is done, we just need to sort the array and output the results in the new order of the objects in the array. Here's the code:

```
for (int loop=1; loop<=numCases; loop++) {

    int n = fin.nextInt();
    student[] group = new student[n];
    for (int i=0; i<n; i++)
        group[i] = new student(fin.next());

    Arrays.sort(group);

    System.out.println("Class #"+loop+" ordering");
    for (int i=0; i<n; i++)
        System.out.println(group[i]);
    System.out.println();
}
```

Generally speaking, this solution is really clean in that each part does it's job and no single part of the code is unnecessarily complicated.

On the next page we'll have our third example.

Weather Data

Many website post data publicly for various reasons. Once we can read text files, we can write custom programs to process that data. In this example, we'll show some data from a website that used to exist. The site used to post files for various cities around the world, with an entry of the average temperature on each day in that city (in Fahrenheit). If, for some reason, there was no reading on a day, then in their file they would store -99. (A temperature that doesn't occur on Earth!)

In this example, we'll use the file FLORLAND.txt which stores daily temperatures in Orlando, Florida from 2005 through most of 2018. The file format is as follows:

Each line of the file has four tab separated values:

Month, Day, Year, Temperature

The first three are integers while the last one is a floating point number specified to 1 decimal place. The days in the file are listed in chronological order.

We'll has the hasNextLine() method to detect when we've reached the end of the file.

To keep this example relatively simple, we'll do the following:

1) Calculate the average temperature by year
2) Calculate the average temperature by month

We'll use arrays to do both. We'll use two arrays for each task. One will store the sum of the readings for the appropriate year or month and the other will store the number of readings for the appropriate year or month.

To have a good object-oriented design, we'll create a city class that stores the four arrays in question. The constructor will take in a Scanner reference, which needs to be pointing to the beginning of the weather file. The constructor will then read through all of the data and populate the its arrays. The only other method the class will have is a toString method. This method will create a String representation of the object, which will be the two charts in question.

Let's look at the constants and instance variables first:

```
final private static int TOTALYEARS = 2027;
final private static int NUMMONTHS = 12;

private double[] yearSum;
private int[] yearNumReadings;
private double[] monthSum;
private int[] monthNumReadings;
```

Now, let's look at the constructor on the next page. It's job is first to allocate arrays, then to loop through all the data in the file, updating the arrays:

```
public city(Scanner fin) {

    yearSum = new double[TOTALYEARS];
    yearNumReadings = new int[TOTALYEARS];
    monthSum = new double[NUMMONTHS+1];
    monthNumReadings = new int[NUMMONTHS+1];

    while (fin.hasNextLine()) {

        StringTokenizer tok = new StringTokenizer(fin.nextLine
        int month = Integer.parseInt(tok.nextToken());
        int day = Integer.parseInt(tok.nextToken());
        int year = Integer.parseInt(tok.nextToken());
        double temp = Double.parseDouble(tok.nextToken());

        yearSum[year] += temp;
        monthSum[month] += temp;
        yearNumReadings[year]++;
        monthNumReadings[month]++;
    }
}
```

This code is very clearly in three sections. Before the loop we allocate our arrays. In the loop the first half of the code reads in the raw data for one line. The second half of the code updates each of our arrays accordingly.

Now, let's shift our attention to the toString() method. This has to build a large string, storing two charts. If we wanted it to be more efficient, we would use a mutable object such as a StringBuffer, but since we don't expect this string to be hundreds of thousands of characters long, this will suffice.

We will just concatenate onto the String, instead of printing out. Here is the code:

```
public String toString() {

    String chart = "Year\tAvg Temp\n";
    for (int i=0; i<yearSum.length; i++) {
        if (yearNumReadings[i] == 0) continue;
        chart = chart + i + "\t" + (yearSum[i]/yearNumReadings[i]) + "\n";
    }

    chart = chart + "\n\n";
    chart = chart + "Month\tAvg Temp\n";

    for (int i=1; i<=NUMMONTHS; i++) {
        if (monthNumReadings[i] == 0) continue;
        chart = chart + i + "\t" + (monthSum[i]/monthNumReadings[i]) + "\n";
    }

    return chart;
}
```

Again, our structure is quite clean. We start our string off with a header, then concatenate into it, each row of the year chart. This is followed by concatenating in some separation and another header, followed by a very similar loop for the months.

Here is the output this program produces when given the file FLORLAND.txt posted online:

```
Year  Avg Temp
1995  71.48630136986296
1996  69.54426229508196
1997  71.75095890410964
1998  71.06739726027396
1999  71.19835616438351
2000  71.15819672131151
2001  71.26109589041099
2002  69.87205479452057
2003  71.58273972602744
2004  71.72759562841532
2005  71.56493150684932
2006  72.3115068493151
2007  72.06410958904105
2008  66.31721311475417
2009  71.52438356164386
2010  69.5320547945206
2011  72.77452054794517
2012  72.22103825136611
2013  72.68602739726022
2014  72.01041095890407
2015  74.23808219178089
2016  72.97404371584697
2017  72.84630136986301
2018  73.74392857142851


Month Avg Temp
1     59.617204301075326
2     62.825811209439564
3     66.8145161290323
4     71.53888888888898
5     76.61344086021508
6     77.84805555555558
7     80.88212365591399
8     80.73548387096778
9     79.26833333333335
10    74.4369444444444
11    65.66028985507243
12    61.13702664796636
```