CSC 1051 Algorithms & Data Structures I

File Input & Output

FILE INPUT & OUTPUT

Reading and Writing Text Files

A Scanner object can be set up to read from various sources, including a text file

Instead of create a Scanner that reads from System.in, it can be sent a File object

```
File file = new File("input.txt");
Scanner in = new Scanner(file);
```

The File class is part of the java.io package

The scanner's hasNext method (or some variation) is usually used to determine when the end of the file is encountered

Reading and Writing Text Files

Creating a Scanner object from a File will cause a FileNotFoundException to be thrown if the file doesn't exist

This is a checked exception, which means it must be dealt with one way or another

It could be caught and handled explicitly

Or a throws clause could be added to the method header to indicate that the method may throw that exception

Reading and Writing Text Files

This program reads an input file one line at a time and prints it to standard output (the console window)

```
public static void main(String[] args) throws
    FileNotFoundException
{
    Scanner in = new Scanner(new File("inDemo.txt"));

    String line;

    while (in.hasNextLine())
    {
        line = in.nextLine();
        System.out.println(line);
    }
}
```

Reading and Writing Text Files

It’s easy to forget to create a File object when setting up a Scanner:

```
Scanner in = new Scanner(“input.txt”);
```

This creates a valid Scanner object, but not one that reads from a file

A Scanner can be created to parse a String, too

Reading and Writing Text Files

A PrintWriter object can be used to write to an output file

```
PrintWriter out = new PrintWriter("output.txt");
```

PrintWriter is also part of the java.io package

Note that no File object is needed

If there is no such file, it is created

If the file exists, the current contents will be overwritten and lost

The PrintWriter constructor may throw an IOException if there is any problem creating the file

Reading and Writing Text Files

Like System.out, a PrintWriter object has print, println, and printf methods that can be used to write output to the file

Output files should be closed using the close method when you're done writing to them

If you don't, data might be lost or corrupted

Reading and Writing Text Files

This program writes some lyrics to an output file:

```
public static void main(String[] args) throws IOException
{
    PrintWriter out = new PrintWriter("outDemo.txt");

    out.println("You say you want a revolution");
    out.println("Well, you know");
    out.println("We all want to change the world");
    out.println();
    out.println("You tell me that it's evolution");
    out.println("Well, you know");
    out.println("We all want to change the world");

    out.close();
}
```

Reading and Writing Text Files

This program reads an input file and echoes it to an output file with line numbers added:

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

public class LineNumbers
{
    public static void main(String[] args) throws IOException
    {
        Scanner in = new Scanner(new File("poem.txt"));
        PrintWriter out = new
            PrintWriter("poemWithLineNumbers.txt");

        String line;

...
```

Reading and Writing Text Files

```
        int i = 1;
        while (in.hasNextLine())
        {
            line = in.nextLine();

            if (line.equals(""))  // don't number blank lines
                out.println();
            else
            {
                out.print(i + "\t");
                out.println(line);
                i = i + 1;
            }
        }

        out.close();
    }
}
```

Reading and Writing Text Files

Input File

Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;

Then took the other, as just as fair,
And having perhaps the better claim,
Because it was grassy and wanted wear;
Though as for that the passing there
Had worn them really about the same,

. . .

Output File

1 Two roads diverged in a yellow wood,
2 And sorry I could not travel both
3 And be one traveler, long I stood
4 And looked down one as far as I could
5 To where it bent in the undergrowth;

6 Then took the other, as just as fair,
7 And having perhaps the better claim,
8 Because it was grassy and wanted wear;
9 Though as for that the passing there
10 Had worn them really about the same,

. . .

Example: Counting Numbers

The field of data analytics, and related areas, often need to read values from a file, process them in some way or perform calculations with them, and output the some results.

In Java, the Scanner class provide a convenient interface for reading specific values, both interactively and from a file. The hasNext and next methods handle checking for and reading values of various types.

This example demonstrates reading values from a text file and analyzing those values to find a minimum, maximum, and average of those values.

Example: Counting Numbers

Using a FileReader and Scanner, values are read one at a time from a large file of integers, and counted, summed, and their minimum and maximum values determined.

```
FileReader input = new FileReader("numbers.txt");
Scanner scan = new Scanner(input);
double total = 0, int count = 0;
int minimum = Integer.MAX_VALUE, minimum = Integer.MIN_VALUE;

while (scan.hasNextInt())
{
    int number = scan.nextInt();
    count++;
    total += number;
    if (number < minimum)
        minimum = number;
    if (number > maximum)
        maximum = number;
}
```

Example: Counting Numbers

Once a total, count, minimum, and maximum are found, results can be printed:

```
System.out.println("Total: " + total);
System.out.println("Count: " + count);
System.out.println("Minimum: " + minimum);
System.out.println("Maximum: " + maximum);
System.out.println("Average: " + total / count);
```

```
Total: 505042.0
Count: 10000
Minimum: 1
Maximum: 100
Average: 50.5042
```

Buffered Streams

Interactive input with user-entered strings or integers is often done with the convenience of the Scanner class.

When reading a lot of data from a file, a buffered stream can be more efficient because it can *buffer* the data by reading thousands of bytes ahead.

Without a buffer, each read requires the operating system to get involved with each read and write operation, which can slow down input and output a lot!

Buffered Streams

A BufferedReader object can be constructed using a FileReader object.

The FileReader handles low level reads and writes while the BufferedReader reduces how many of those operations are needed by buffering the data.

```
FileReader input = new FileReader("hugedatafile.txt");

BufferedReader reader = new BufferedReader(input);
```

Buffered Streams

This example code reads the contents of a text file and prints it out to the screen, line by line.

```
FileReader input = new FileReader("hugedatafile.txt");
BufferedReader reader = new BufferedReader(input);

String line;
while ((line = reader.readLine()) != null)
{
	System.out.println(line);
}

reader.close();
```

It is good practice to close the buffered stream when you're done with it to avoid losing data.

Buffered Streams

A BufferedReader and BufferedWriter can work together to efficiently copy a file, like this:

```
FileReader input = new FileReader("originalfile.txt");
BufferedReader reader = new BufferedReader(input);

FileWriter output = new FileWriter("copiedfile.txt");
BufferedWriter writer = new BufferedWriter(output);

String line;
while ((line = reader.readLine()) != null)
{
	writer.write(line + "\n"); // readLine removes
newlines
}

reader.close();
writer.close();
```

Buffered Streams

The FileReader, FileWriter, BufferedReader, and BufferedWriter classes are best for textual data. For binary data, including images, Java provides these corresponding classes:

- FileInputStream – reads binary data
- BufferedInputStream – buffers binary data that is read
- FileOutputStream – writes binary data
- BufferedOutputStream – buffers binary data to write

Buffered Streams

This example shows how to copy a binary file:

```
FileInputStream input = new FileInputStream("orig.jpg");
BufferedInputStream reader = new BufferedInputStream(input);

FileOutputStream output = new FileOutputStream("copy.jpg");
BufferedOutputStream writer = new BufferedOutputStream(output);

byte[] buffer new byte[4096];

while ((int numBytes = reader.read(buffer)) != -1)
{
        writer.write(buffer, 0, numBytes); newlines
}

reader.close();
writer.close();
```

You *could* copy a text file this way, too, since a text file is a special case of a binary file. The opposite is not true, though.

Example: Counting Letters

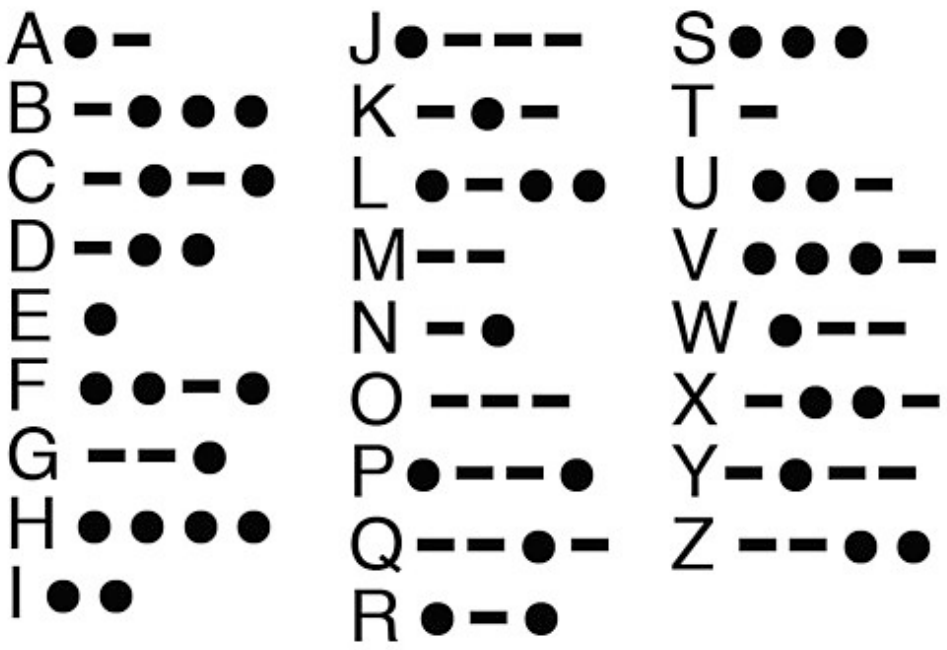
Counting the occurrences of data is an important concept in science. The frequency with which a specific data item occurs can be used for encryption, data compression, code cracking, cybersecurity, and text mining.

One of the earliest uses of data frequency is Morse Code, which was developed by Samuel Morse and first used in 1844 to efficiently communicate text information with a telegraph.

Example: Counting Letters

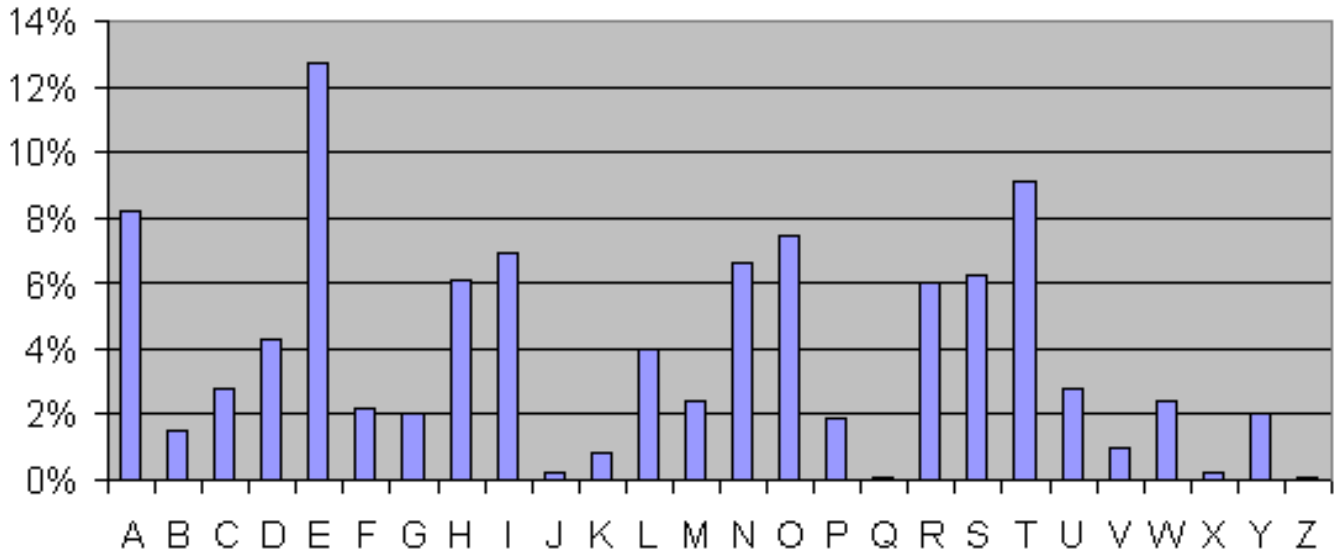
In Morse Code, each letter of the alphabet is made up of combinations of dots and dashes. The more common a letter is, the fewer dots and dashes are used for that letter.

Morse Code Alphabet



Example: Counting Letters

The frequency of letters occurring in written English language was used to create the Morse Code alphabet. Here's a general idea of how common letters in English are:



The creation of Morse Code was an early use of data mining.

Example: Counting Letters

A map can be used to count letters. The for loop iterates through the characters, incrementing its entry in counts.

```
Stream text = "I have a dream that my four little children " +
    "will one day live in a nation where they will not be " +
    "judged by the color of their skin but by the content " +
    "of their character.";

Map counts = new LinkedHashMap();

for (int i = 0; i < text.length(); i++)
{
    char letter = text.charAt(i);
    if (Character.isAlphabetic(letter))
    {
        if (counts.containsKey(letter))
            counts.put(letter, counts.get(letter) + 1);
        else
            counts.put(letter, 1);
    }
}
```

Example: Counting Letters

The entries in the map are printed using a for-each loop.

```
System.out.println("Letter frequencies:");

boolean first = true;
for (Character letter : counts.keySet())
{
    System.out.print((first ? "" : ", ") + letter +
                     ": " + counts.get(letter));
    first = false;
}
```

```
Letter frequencies:
I: 11, H: 10, A: 9, V: 2, E: 17, D: 5, R: 9, M: 2, T: 15, Y: 5, F: 3, O:
9, U: 3, L: 9, C: 5, N: 9, W: 3, B: 4, J: 1, G: 1, S: 1, K: 1
```

Example: Counting Letters

Here is the result of counting letters in Dr. Martin Luther King, Jr.'s famous "I Have a Dream" speech:

```
A  ######################################## 561 (7.8%)
B  ######## 114 (1.6%)
C  ############ 183 (2.5%)
D  ################## 257 (3.6%)
E  ################################################################ 891 (12.3%)
F  ############### 224 (3.1%)
G  ############ 177 (2.4%)
H  ########################### 389 (5.4%)
I  ######################################## 568 (7.8%)
J  # 22 (0.3%)
K  ### 52 (0.7%)
L  ######################## 341 (4.7%)
M  ############# 190 (2.6%)
N  ################################## 476 (6.6%)
O  ########################################### 608 (8.4%)
P  ###### 97 (1.3%)
Q  # 6 (0.1%)
R  ############################## 419 (5.8%)
S  ############################## 429 (5.9%)
T  ############################################## 672 (9.3%)
U  ############ 176 (2.4%)
V  ##### 82 (1.1%)
W  ############ 164 (2.3%)
X  # 5 (0.1%)
Y  ######### 127 (1.8%)
Z  # 6 (0.1%)
```