Programming Languages and Techniques (CIS120)

Lecture 31

April 6th, 2016

I/O

Chapter 28

Poll

Did you finish HW 07 PennPals?

1. Yes!
2. I turned it in on time, but there are a few things I couldn't figure out
3. I'm planning to use the late period for this assignment

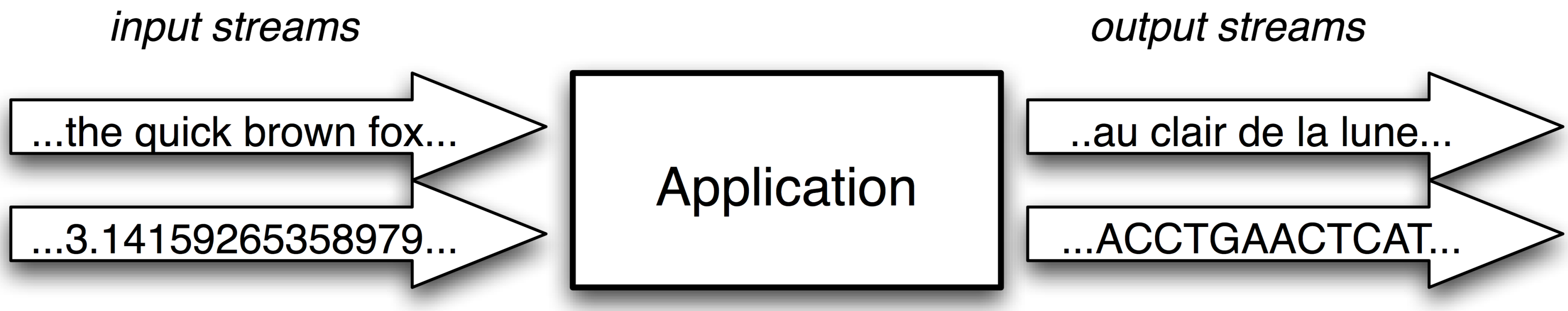
Announcements

- HW8: Spellchecker
  - Available now
  - Due: Tuesday, April 12th at midnight
  - Parsing, working with I/O, more practice with collections

java.io

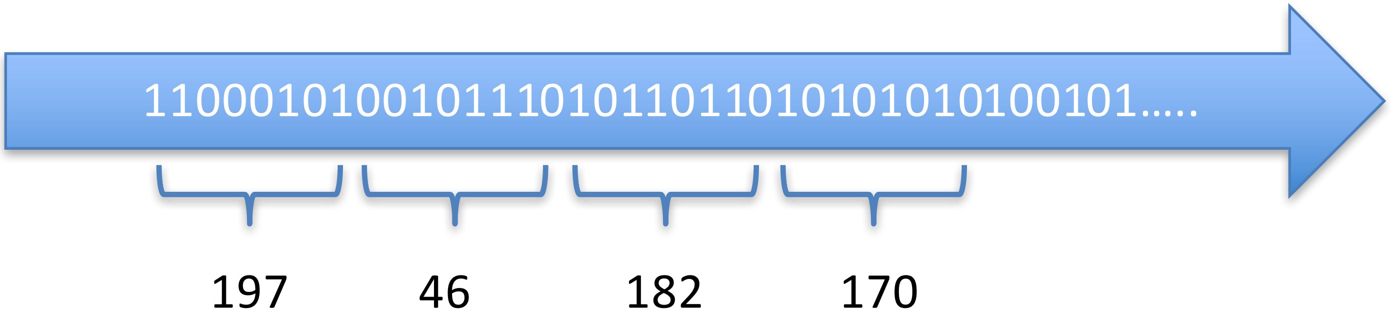
I/O Streams

- The *stream* abstraction represents a communication channel with the outside world.
  - can be used to read or write a potentially unbounded number of data items (unlike a list)
  - data items are read from or written to a stream one at a time
- The Java I/O library uses subtyping to provide a unified view of disparate data sources and sinks.



Low-level Streams

- At the lowest level, a stream is a sequence of binary numbers



11000101001011101011011010101010100101.....

197 46 182 170

- The simplest IO classes break up the sequence into 8-bit chunks, called *bytes*. Each byte corresponds to an integer in the range 0 – 255.

`InputStream` and `OutputStream`

- Abstract classes that provide basic operations for the Stream class hierarchy:

```
int read ();          // Reads the next byte of data
void write (int b); // Writes the byte b to the output
```

- These operations read and write `int` values that represent *bytes*

  range `0–255` represents a byte value

  `–1` represents “no more data” (when returned from read)

- java.io provides many subclasses for various sources/sinks of data:

  files, audio devices, strings, byte arrays, serialized objects

- Subclasses also provides rich functionality:

  encoding, buffering, formatting, filtering

Binary IO example

```
InputStream fin = new FileInputStream(filename);

int[] data = new int[width][height];
for (int i=0; i < data.length; i++) {
   for (int j=0; j < data[0].length; j++) {
      int ch = fin.read();
      if (ch == -1) {
        fin.close();
        throw new IOException("File ended early");
      }
      data[j][i] = ch;
   }
}
fin.close();
```

BufferedInputStream

- Reading one byte at a time can be slow!
- Each time a stream is read there is a fixed overhead, plus time proportional to the number of bytes read.

```
disk -> operating system -> JVM -> program
disk -> operating system -> JVM -> program
disk -> operating system -> JVM -> program
```

- A `BufferedInputStream` presents the same interface to clients, but internally reads many bytes at once into a *buffer* (incurring the fixed overhead only once)

```
disk -> operating system ->>>> JVM -> program
                               JVM -> program
                               JVM -> program
                               JVM -> program
```

Buffering Example

```
FileInputStream fin1 = new FileInputStream(filename);
InputStream fin = new BufferedInputStream(fin1);

int[] data = new int[width][height];
for (int i=0; i < data.length; i++) {
   for (int j=0; j < data[0].length; j++) {
      int ch = fin.read();
      if (ch == -1) {
        fin.close();
        throw new IOException("File ended early");
      }
      data[j][i] = ch;
   }
}
fin.close();
```

`PrintStream` Methods

`PrintStream` adds buffering and conversion methods to `OutputStream`

```
void println(int  i);       //  write i followed by a newline
void println(String  s);    //  write s followed by a newline
void println();             //  write a newline to the stream

void print(String  s);      //  write s without terminating the line
                                (output may not appear until the stream is flushed)
void flush();               //  actually output characters waiting to be sent
```

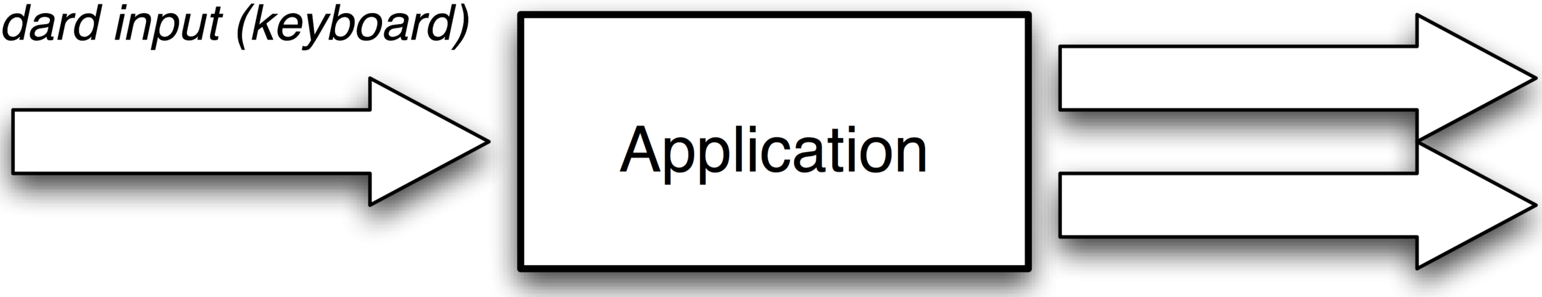
- Note the use of *overloading*: there are *multiple* methods called `println`
  - The compiler figures out which one you mean based on the number of arguments, and/or the *static* type of the argument you pass in at the method’s call site.
  - The java I/O library uses overloading of constructors pervasively to make it easy to “glue together” the right stream processing routines

Output Example

```java
OutputStream out = new FileOutputStream("F");
PrintStream p = new PrintStream(out);
p.println("P5");
p.println("512 512");
p.println("255");
for (int i=0; i<HEIGHT; i++) {
   for (int j=0; j<WIDTH; j++) {
      p.write(data[j][i]);
   }
}
p.close();
```

The Standard Java Streams

java.lang.System provides an `InputStream` and two standard `PrintStream` objects for doing console I/O.



`System.in`

standard input (keyboard)

Application

`System.out`

standard output (display)

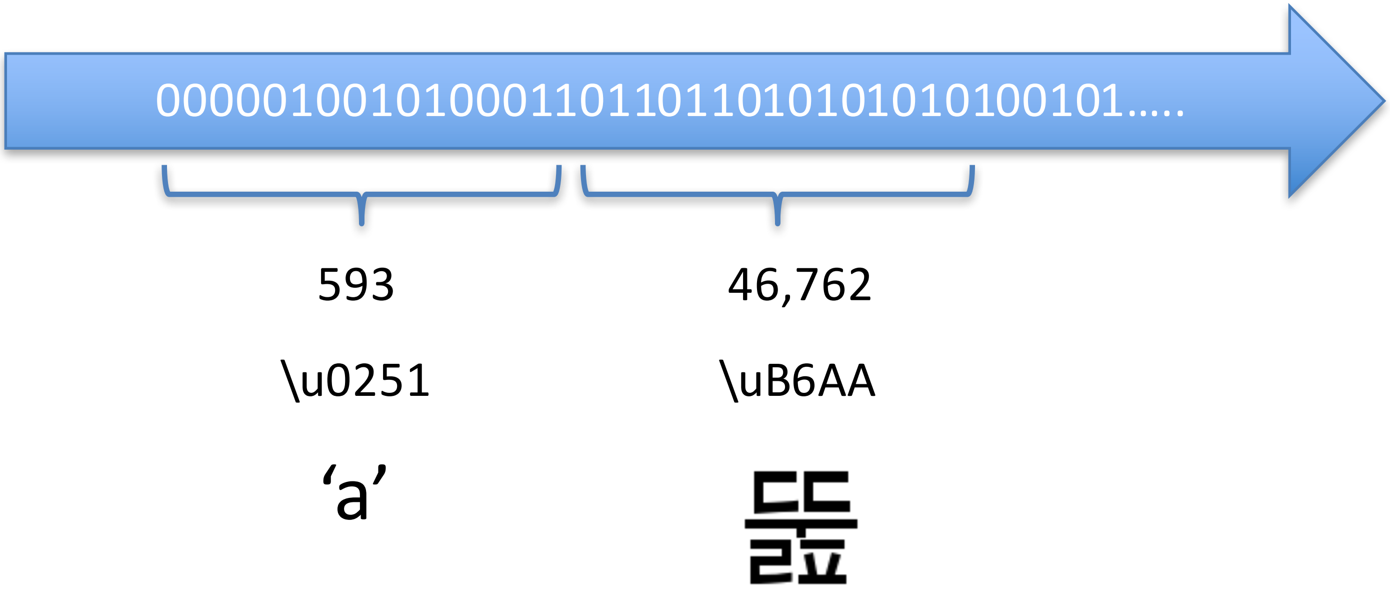
standard error (display)

`System.err`

Note that `System.in`, for example, is a *static member* of the class System – this means that the field “`in`” is associated with the *class*, not an *instance* of the class. Recall that static members in Java act like global variables.

Character based IO

A character stream is a sequence of 16-bit binary numbers



The character-based IO classes break up the sequence into 16-bit chunks, of type `char`. Each character corresponds to a letter (specified by a *character encoding*).

Reader and Writer

- Similar to the `InputStream` and `OutputStream` classes, including:

```
int read ();          // Reads the next character
void write (int b); // Writes the char to the output
```

- These operations read and write `int` values that represent *unicode characters*
  - read returns an integer in the range 0 to 65535 (i.e. 16 bits)
  - value `-1` represents “no more data” (when returned from read)
  - requires an “encoding” (e.g. UTF-8 or UTF-16, set by a `Locale`)
- Like byte streams, the library provides many subclasses of Reader and Writer Subclasses also provides rich functionality.
  - use these for portable text I/O
- Gotcha: `System.in`, `System.out`, `System.err` are *byte* streams
  - So wrap in an InputStreamReader / PrintWriter if you need unicode console I/O

PrintStream vs. Writer

```
PrintStream p = new PrintStream(new FileOutputStream("out1"));
Writer w = new FileWriter("out2");
```

Which of these will produce the same output file?

1.
```
p.print(120);
w.write(120);
```

2.
```
p.print("120");
w.write("120");
```

4. Both

5. None

Answer: 2. (The print(int) method converts ints to text in the first example)

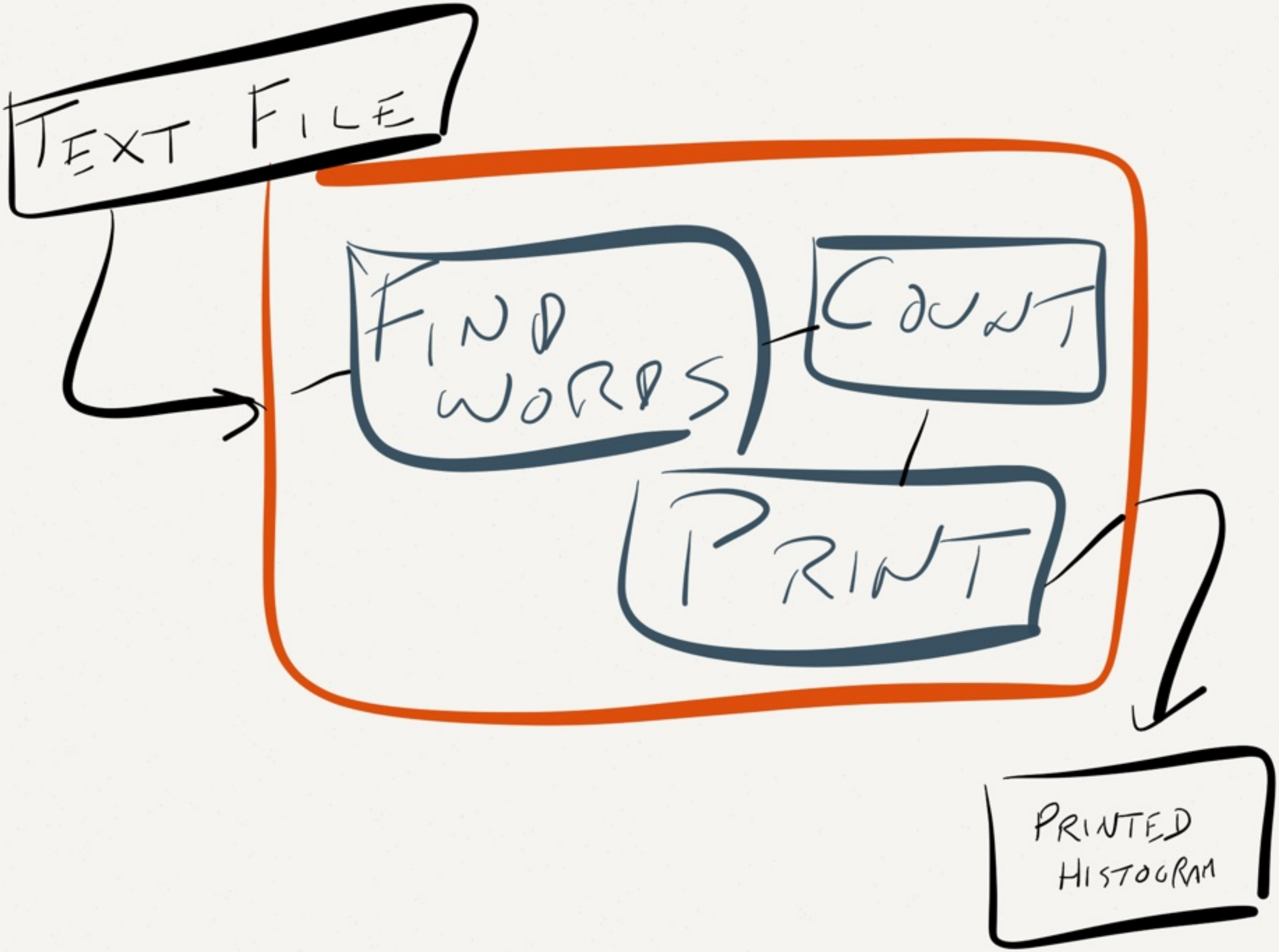
Text IO Example: Histogram.java

A design exercise using java.io and
the generic collection libraries

Problem Statement

Write a program that, given a filename for a text file as input, calculates the frequencies (i.e. number of occurrences) of each distinct word of the file. The program should then print the frequency distribution to the console as a sequence of “word: freq” pairs (one per line).

```
Histogram result:
The : 1
Write : 1
a : 4
as : 2
calculates : 1
command : 1
console : 1
distinct : 1
distribution : 1
e : 1
each : 1
file : 2
filename : 1
for : 1
freq : 1
frequencies : 1
frequency : 1
given : 1
i : 1
input : 1
line : 2
number : 1
occurrences : 1
of : 4
one : 1
pairs : 1
per : 1
print : 1
program : 2
sequence : 1
should : 1
text : 1
that : 1
the : 4
then : 1
to : 1
word : 2
```


TEXT FILE
FIND WORDS
COUNT
PRINT
PRINTED HISTOGRAM

Decompose the problem

- Sub-problems:
    1. How do we iterate through the text file, identifying all of the words?
    2. Once we can produce a stream of words, how do we calculate their frequency?
    3. Once we have calculated the frequencies, how do we print out the result?

- What is the interface between these components?
- Can we test them individually?

Histogram Structure

Which data structure should we use to store the histogram?

1. Set<String>
2. Set<Integer>
3. Map<Integer, String>
4. Map<String,Integer>
5. Map<String,Set<String>>