Programming Languages and Techniques (CIS1200)

Lecture 31

I/O & Histogram Demo

Chapter 28

Announcements

- HW08: ChatterBot
  - Available now; due on Thursday, April 17th
  - Practice with I/O and Collections

- Game Design Proposal Milestone Due: (8 points)
  Tuesday, April 15th at Midnight = 11:59PM!
  - (Should take about 1 hour)
  - Submit on GRADESCOPE
  - TAs will give you feedback

Announcements

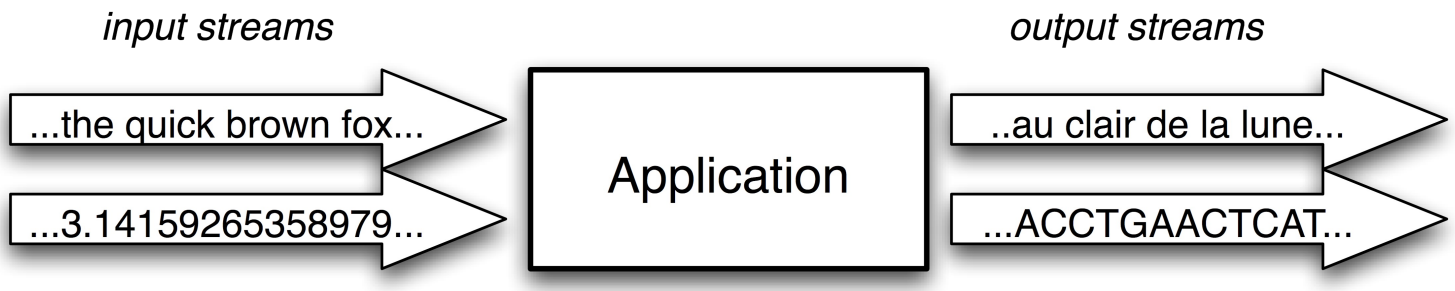
- TA position applications are available
  - CIS 1100, 1200, 1600, 1210 (see link on Ed)
  - Other CIS classes (see https://www.cis.upenn.edu/ta-information/)
  - Accepting applications until Friday, April 18th
  - Intro CIS TA Panel: April 14th 7-8:30pm, Berger Auditorium

Recap: java.io

Viewing sequential data as a stream

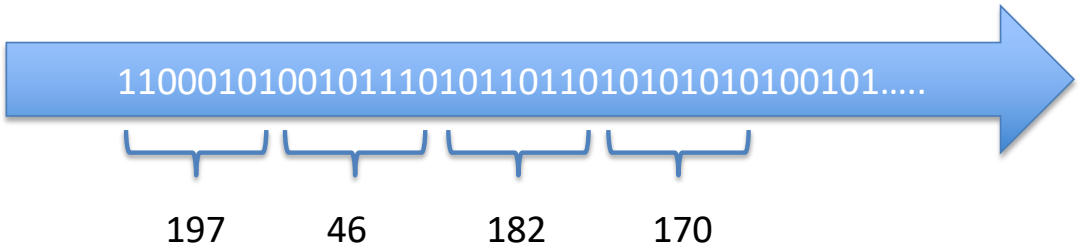
I/O Streams

- 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



- 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.

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

Rule of thumb times to access data:

	actual	for intuition
CPU:	0.5ns	(~ 1 sec)
RAM:	100 ns	(~ 1.6 minutes)
SSD:	150,000 ns	(~ 2.75 days)

*simplified explanation – the OS, disk, etc., might use caching to speed things up

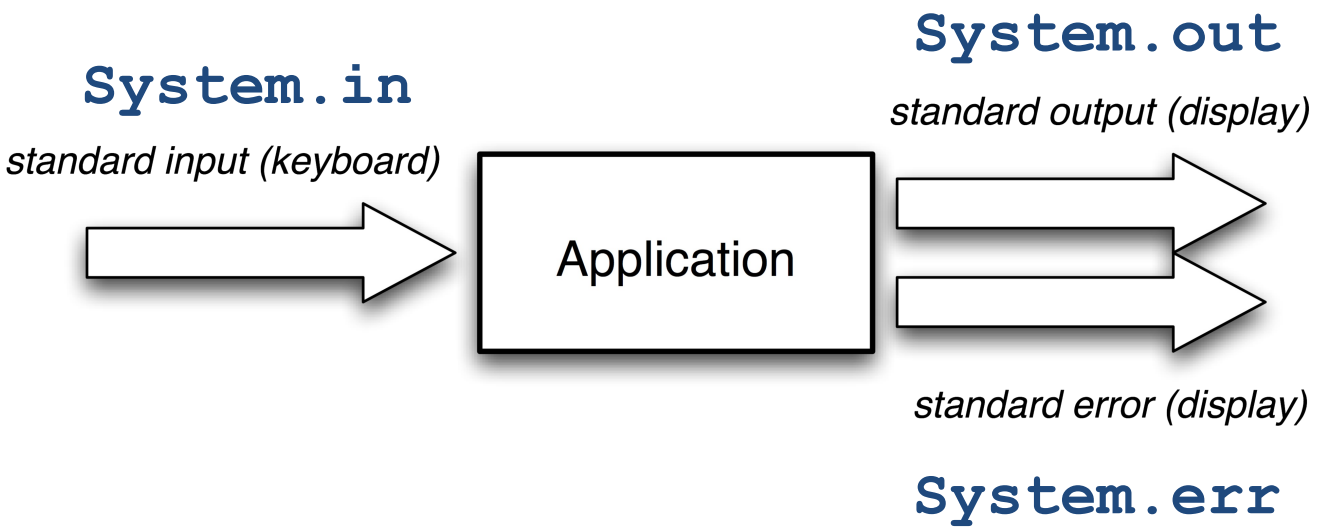
Buffering Example

```
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();
```

The Standard Java Streams

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



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.

`PrintStream` Methods

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

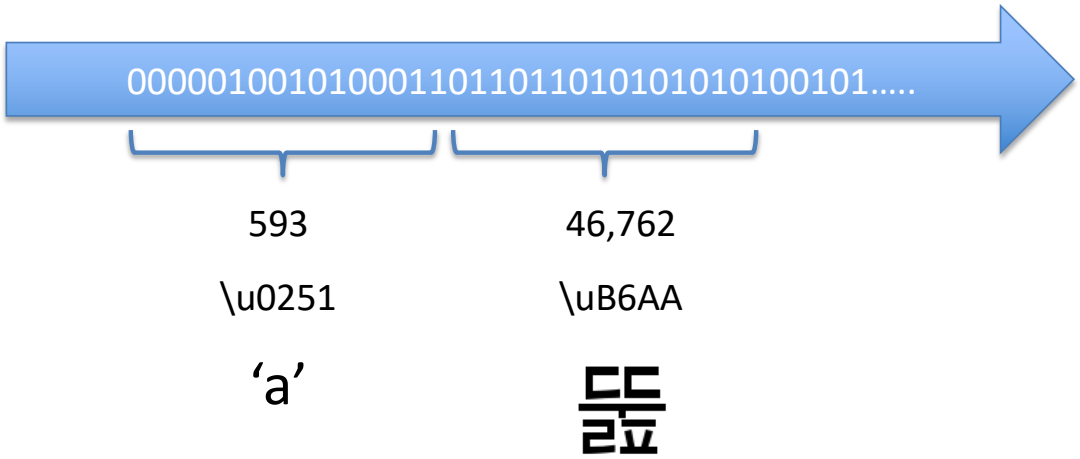
```
void println(boolean b); // write b followed by a new line
void println(String s);  // write s followed by a newline
void println();          // write a newline to the stream

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

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

Design Example: Histogram.java

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

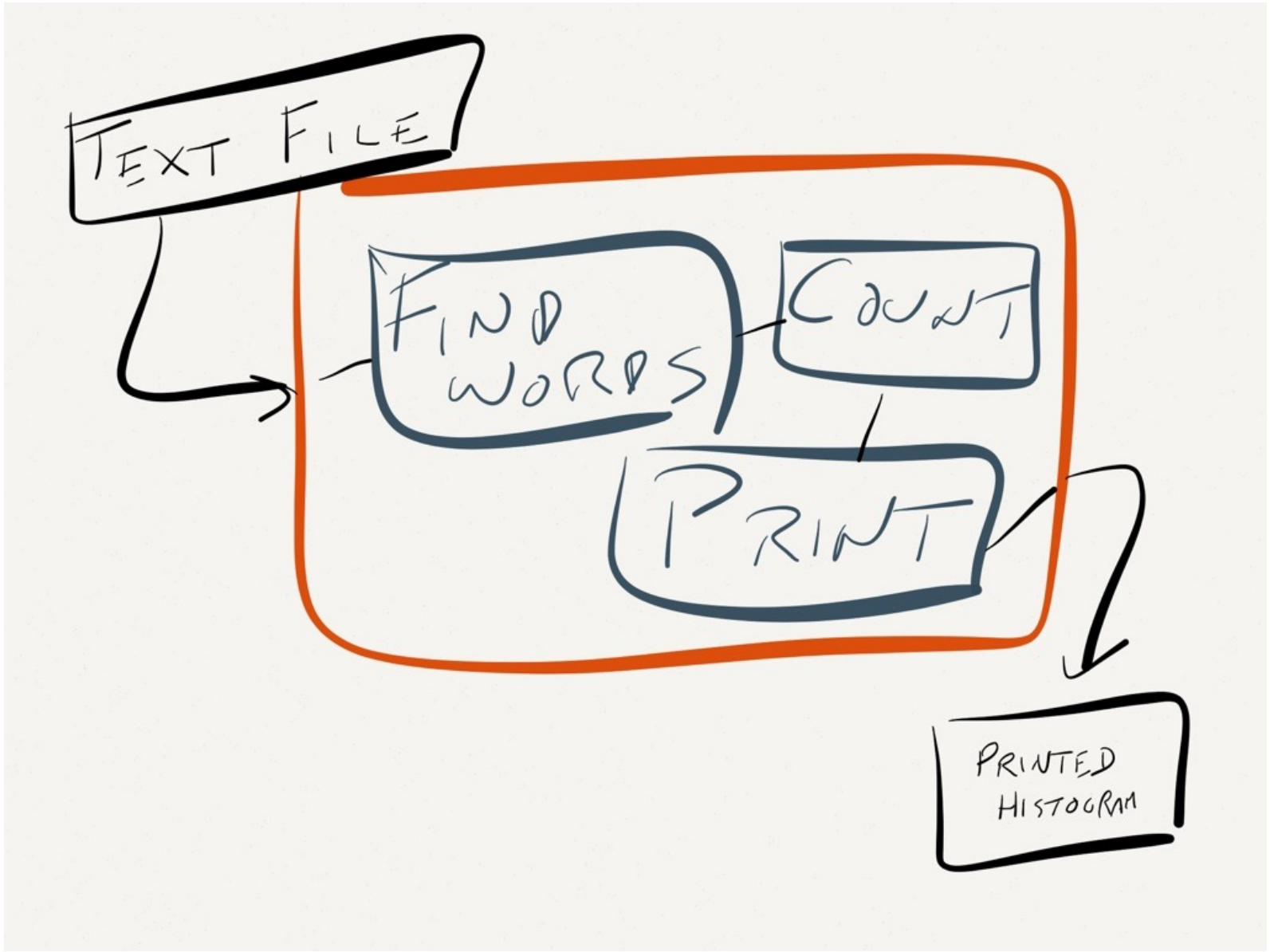
(SEE COURSE NOTES FOR THE FULL STORY)

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?

How to produce a stream of words?

1. How do we iterate through the text file, identifying all of the words?

```
public interface Iterator<T> {
    // returns true if the iteration has more elements
    public boolean hasNext();
    // returns the next element in the iteration
    public T next();
    // Optional: removes last element returned
    public void remove();
}
```

- **Key idea:** Define a class (WordScanner) that implements this interface by reading words from a text file.

Coding: Histogram.java

WordScanner.java

Histogram.java

Iterator – hasNext() – First Attempt?

```
@Override
public boolean hasNext() {
    boolean value = true;
    try {
        int c = r.read();
        if (c == -1) {
            value = false;
        }
    } catch (IOException io) {
        System.out.println("IO Exception happened");
    }
    return value;
}
```

32: Which combination of the following properties form a useful invariant for the WordScanner fields?

0

```
public class WordScanner implements Iterator<String> {
    private Reader r;
    private int c = -1;
    // ...
}
```

Which combination of the following properties form a useful invariant for the WordScanner fields?

1. r is not null
2. r is null if and only if there is no next word

A. c is 0 if there is no next word and nonzero otherwise
B. c is -1 if there is no next word and contains the first character of the next word otherwise

1 & A 0%

1 & B 0%

2 & A 0%

2 & B 0%

```
public class WordScanner implements Iterator<String> {
        private Reader r;
        private int c = -1;
        // ...
}
```

Which combination of the following properties form a useful invariant for the WordScanner fields?

1. r is not null
2. r is null if and only if there is no next word

A. c is 0 if there is no next word and nonzero otherwise
B. c is -1 if there is no next word and contains the first character of the next word otherwise

ANSWER: 1 & B