Programming Languages and Techniques (CIS120)

Lecture 30

April 2, 2012

Streams & IO

Announcements

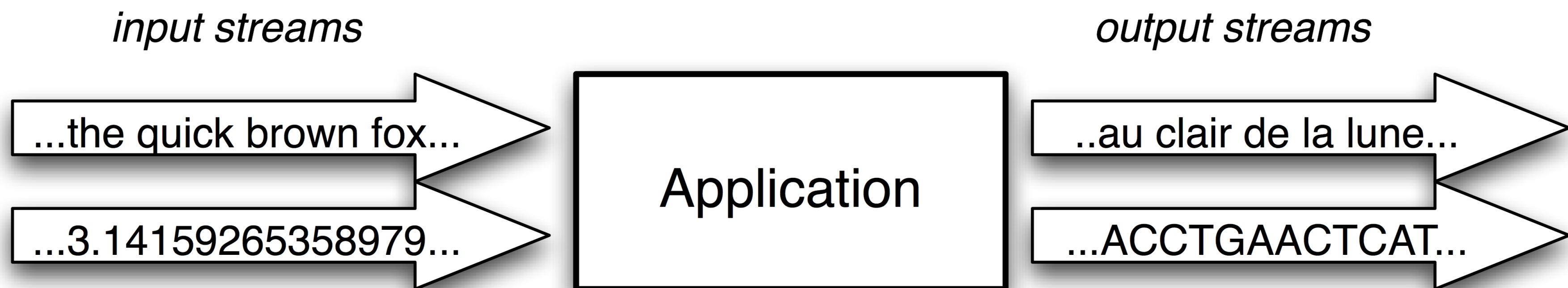
- HW 09 will be available today
  - Covers Java libraries: collections & IO
  - Last automatically graded HW assignment
  - ONLY TWO SUBMISSIONS this time
- Midterm grades will be available soon.
  - *After* the make-up exam. I will post to Piazza when they are ready.

java.io

1. I/O streams
2. Readers & Writers

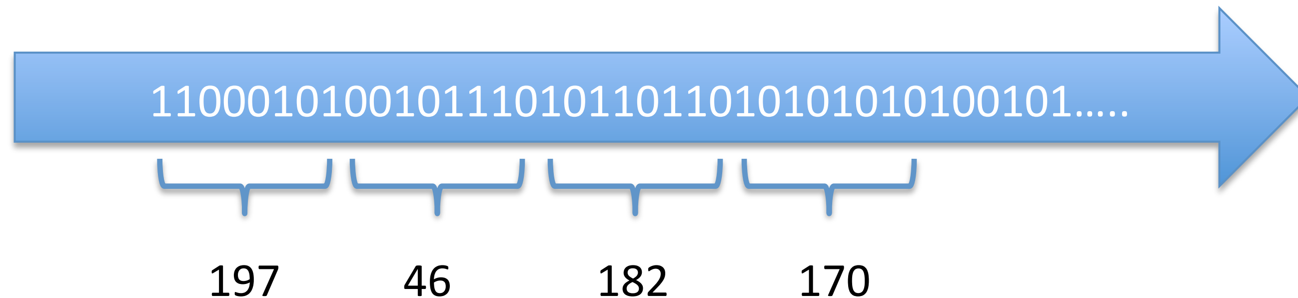
I/O Streams

- The *stream* abstraction represents a communication channel with the outside world.
  - potentially unbounded number of inputs or outputs (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 or data sinks.



Low-level IO

- A stream is a sequence of binary numbers



- The simplest IO classes break up the sequence into 8-bits 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:

```
abstract int read ();          // Reads the next byte of data
abstract 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
  - 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

*Abstract classes are classes that cannot be directly instantiated (via `new`). Instead, they provide partial, concrete implementations of some operations. In this way, abstract classes are a bit like interfaces (they provide a partial specification) but also a bit like classes (they provide some implementation). They are most useful in building big libraries, which is why we aren't focusing on them in this course.

Binary IO example

```
public Image() throws IOException {
   InputStream fin = new FileInputStream(“mandrill.pgm”);

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

BufferedInput Stream

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

- A BufferedInput Stream reads many bytes at once into a buffer (incurring the fixed overhead only once) while still producing the data with the same interface.

Buffering example

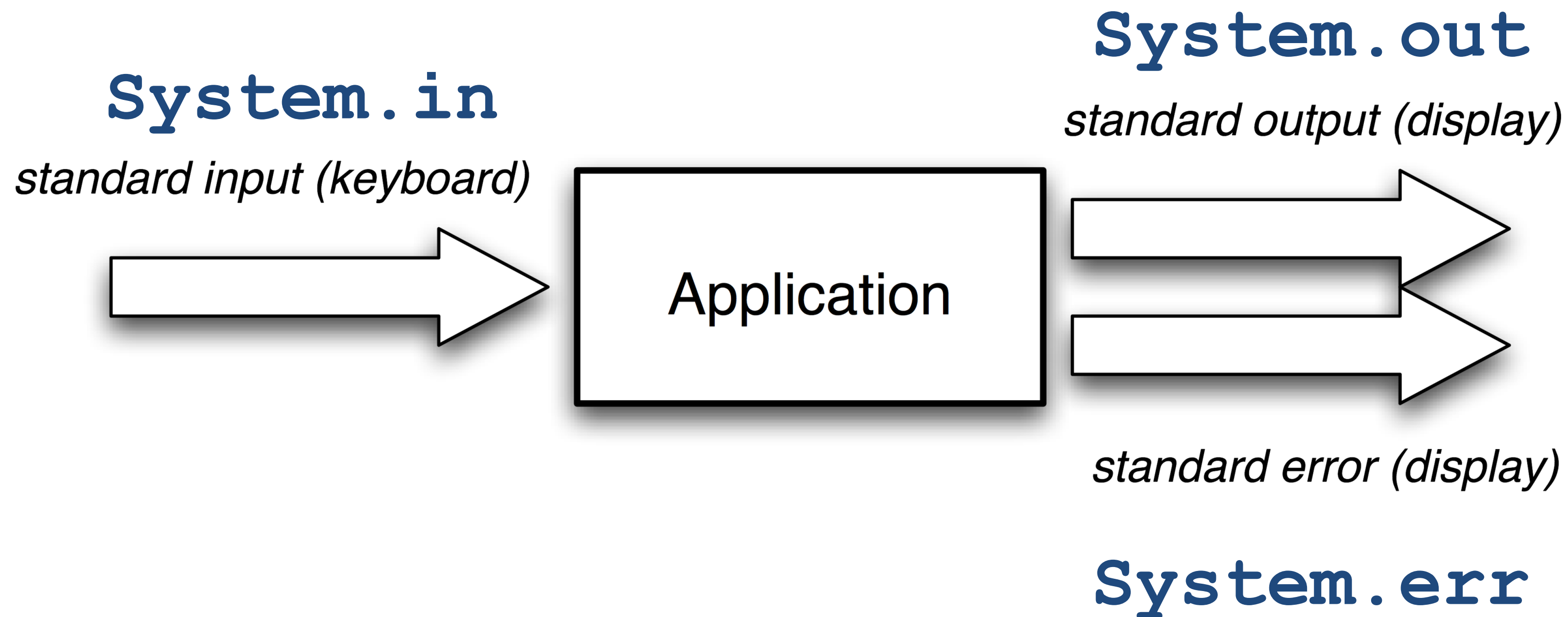
```
public Image() throws IOException {
   FileInputStream fin1 = new FileInputStream("mandrill.pgm");
   InputStream fin = new BufferedInputStream(fin1);

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

Demo

Binary input demo

The Standard Java Streams

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



Note that `System.in` 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. Methods can also be static – the most common being "`main`", but see also the `Math` class.

Example `PrintStream` Methods

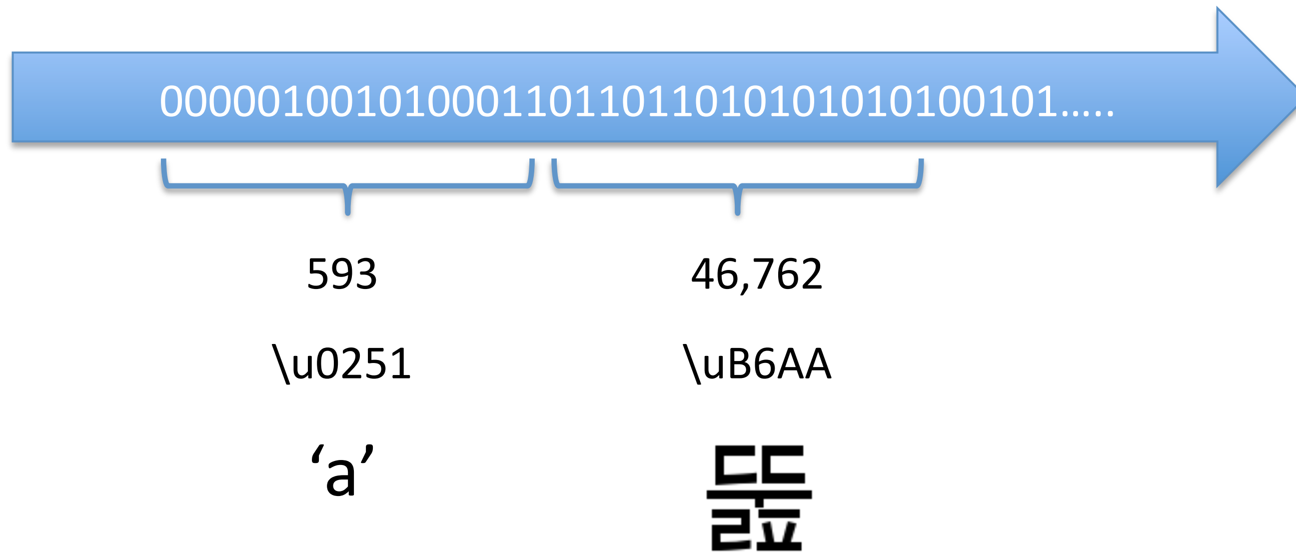
- Adds Buffering and binary-conversion methods to OutputStreams

```
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 any 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 stream is a sequence of binary numbers



- The text-based IO classes break up the sequence into 16-bits chunks, called chars. Each character corresponds to a letter (specified by a character-encoding).

Reader and Writer

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

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

- These operations read and write `int` values that represent *unicode characters*
  - value `-1 still` represents "no more data" (when returned from read)
  - other values need an "encoding" (e.g. UTF-8 or UTF-16, set by a `Locale`)

- As for 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

Demo

How do you read from a file into a String?

FileReadingTest.java

Java I/O Design Strategy Summary

1. Understand the concepts and how they relate:
   - What kind of stream data are you working with?
   - Is it byte-oriented or text-oriented?
     - InputStream vs. InputReader
   - What is the source of the data?
     - e.g. file, console, network, internal buffer or array
   - Does the data have any particular format?
     - e.g. comma-separated values, line-oriented, numeric
     - Consider using Scanner or another parser
2. Design the interface:
   - Browse through java.io libraries (to remind yourself what's there!)
   - Determine how to compose the functionality your need from the library
   - Some data formats require more complex *parsing* to convert the data stream into a useable structure in memory

Design Example: Histogram.java

A design exercise using java.io and generic collections libraries.

Problem Statement

- Write a command-line 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

Interactive Demo

Histogram.java and WordScanner.java