Programming Languages and Techniques (CIS120)

Lecture 29

March 27, 2013

IO

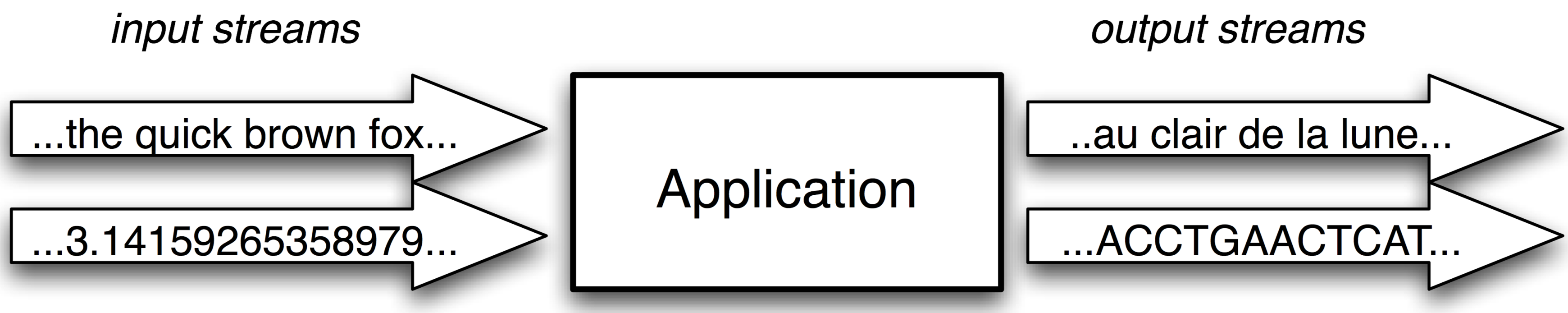
Announcements

- Midterm 2 is Friday
  - Towne 100 last names A—K
  - Cohen G17 last names L—Z
- Review session: Wednesday 6:30-9:30pm
  - Wu & Chen (Levine 101)
  - Lab this week is review (bring questions!)

java.io

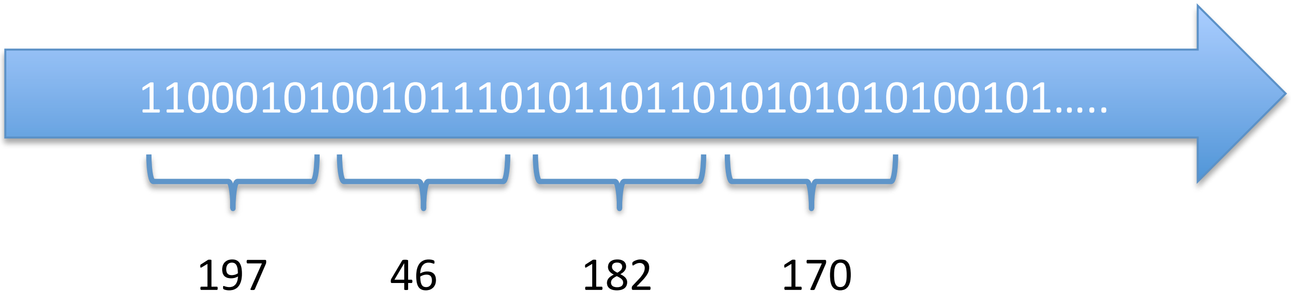
I/O Streams

- 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.



Binary-based IO

- A stream is a sequence of binary numbers



11000101001011101011011010101010100101.....

197 46 182 170

- 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.

Demo

Binary input demo

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) {
                fin.close();
                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.

```
disk -> JVM -> program
disk -> JVM -> program
disk -> JVM -> program
```

- 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.

```
disk ->>>> JVM -> program
           JVM -> program
           JVM -> program
           JVM -> program
```

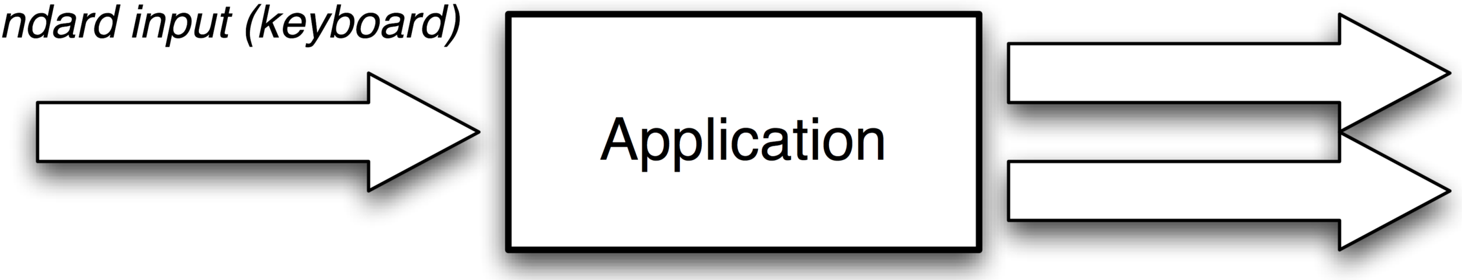
Buffering example

```
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();
}
```

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` 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

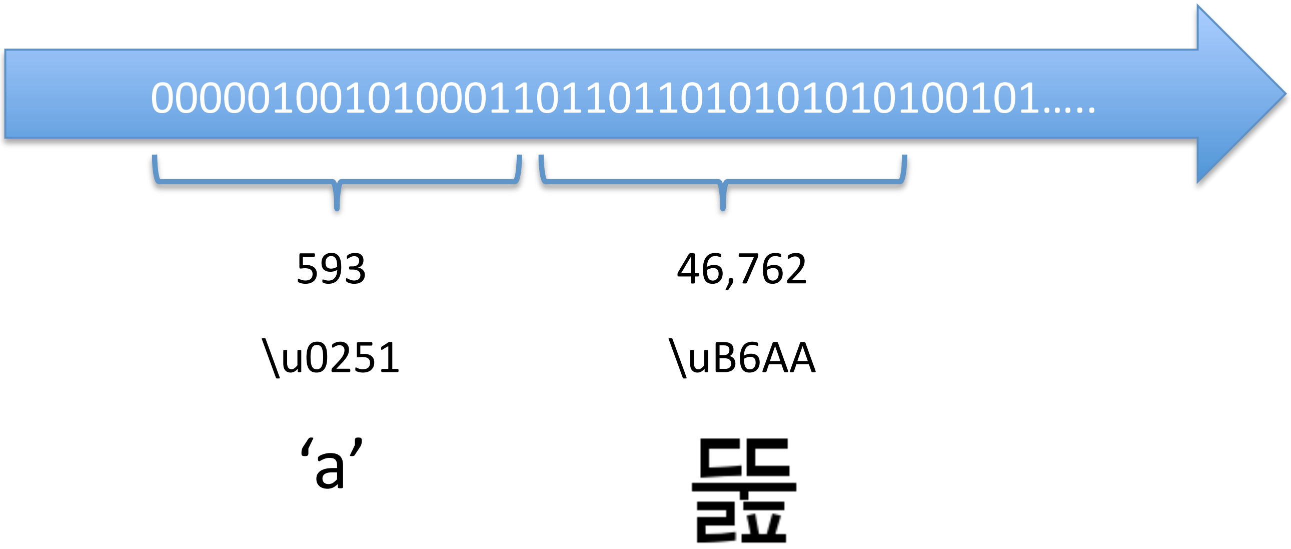
```
void println(boolean b);   // write b followed by a new line
void println(String s);    //  write s followed by a newline
void println();            //  write a newline to the stream

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 character-based IO classes break up the sequence into 16-bit 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 character
abstract 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

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