Programming Languages and Techniques (CIS120e)

Lecture 23

Nov. 3, 2010

I/O Libraries and Java Pragmatics by Example
• Homework 7 is due tonight!

• Homework 8 (SpellChecker) is due next Weds., Nov. 10\textsuperscript{th}.
  – It will be available soon.

• Midterm 2 is Friday, November 12\textsuperscript{th}
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).
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 inheritance to provide a unified view of disparate data sources or data sinks.

*input streams*

...the quick brown fox...

...3.14159265358979...

*output streams*

..au clair de la lune...

...ACCTGAACCTCAT...
InputStream and OutputStream

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

```java
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.
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)

**System.out**
standard output (display)

**System.err**
standard error (display)

Note that `System.in` is a so-called static member of the class `System` – this means that the field “in” is associated with the class, not an instance of the class. 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

```java
void println(String s);  // write s followed by 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 println();  // write a newline to the stream
void flush();     // actually output any characters waiting to be sent
```

- Note the use of overloading: there are two 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
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 character b

• 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 them in a PrintReader / PrintWriter if you need unicode console I/O
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
Interactive Demo

Histogram.java and Tokenizer.java
Java Pragmatics Cheat Sheet

• Program entry point: `public static void main(String[] args)`
  – Command-line arguments are passed in the String array given to main.

• Generic types cannot be instantiated by primitive datatypes (e.g. int, boolean); instead you must use “wrapper” classes (e.g. Integer, Boolean)
  – Java will automatically convert primitive values to wrapped objects.
  – See `java.lang.Integer, java.lang.Character`
  – This is a “kludge” due to Java’s history; generics weren’t added until long after the Java virtual machine was standardized...

• When creating an object of generic type, don’t forget to give type parameters: e.g. `new TreeMap<String, Integer>()`
Java Pragmatics Cheat Sheet

• Static fields and methods are “global” variables attached to a class name.
  – e.g. Character.isLetter(int c)

• Classes can be nested: e.g. Map.Entry<K,V>

• Abstract classes can’t be instantiated, but they make good types.
  – Libraries use abstract classes to encapsulate shared algorithms.

• Calls to overloaded methods and constructors are determined by the number of arguments and their static types.

• Many I/O methods can fail by throwing an exception.
  – Exceptions are for unusual situations: File does not exist, Disk is full, etc.
  – Code that calls such methods can handle the error using:
    try {...} catch (Exception e) {...}