PURDUE UNIVERSITY

CS18000: Problem Solving And Object-Oriented Programming

File I/O

18 April 2011

Prof. Chris Clifton

COMPUTER SCIENCES

Goal: Make Data Useful Beyond Program Execution

- Program data stored in variables

  Okay, a little more than that

  – Arrays

  – Linked data structures
- "Disappear" when program exits

  How do we keep values around?

  – Display in a GUI and write them down?

Idea 1: Persistent Variables

- "Save" value of variables when program exits
  - "Load them back" when program starts
- Advantage: Fits nicely with the way we think of programming
  - Changes concept of initialization
- Disadvantages
  - Which variables saved?
  - What if we want to use data in a different program?

Idea 2: Program-independent storage

- Save desired values in a known format
  - list
  - table
  - …
- Load back into variables when needed
  - Program knows format
  - Gets appropriate values to go into desired variables

Idea 3: A mix of both

- Save in known format
  - Don't directly map values to named variables
- But known only to your own program
  - Proprietary formats (e.g., Excel, Word)
- *We'll stay away from this approach*

Basic idea: *Stream* abstraction

- Values written sequentially
  - Write the value of *name*
  - Write the value of *address*
  - …
  - Write the next *name*
- Read in same order
  - Read *name*
  - Read *address*
  - *…*

Why sequentially?

- History
  - First storage devices were tape drives
  - Inherently sequential
- Does this still make sense?
- Direct Access Storage Device
  - Can move to anyplace
  - But faster in sequence

Stream as an Abstraction

- Many devices good at sequential access
  - Tape drive
  - Telephone line
  - Satellite link
- Fewer good at truly random access
  - Disks read a block, not individual byte
    - Even then, sequential blocks faster
  - USB flash drive seems random access
    - But designed to give a block of sequential data per request
    - Best if you need the whole thing
- *And then there is concurrent access…*

Solution: Stream

- read
  - Get the next value from the stream
- write
  - Put a value into the stream
- skip
  - Skip ahead some number of values
- reset
  - Go back to the beginning
- close
  - Done with this stream

Streams are

- Simple to understand
- Good representations of common I/O devices
  - Disks
  - Networks
  - Speakers
- Computationally sufficient
  - *You'll learn about Turing machines*

Alternative: Avoid the abstraction

- Instead of a Stream, we could use each device using it's own operations. This is:
  A. A good idea, because we get better performance
  B. A good idea, because we might need some special capabilities
  C. A bad idea, because we are supposed to use abstractions
  D. A bad idea, because it locks our program in to a particular device

Java Classes for File I/O

- File
  – Has a name
  – But no operations to get data
- FileOutputStream, FileInputStream
  – Write, read bytes
- Scanner
  – Converts character values to/from primitive types, strings

File class

- Named object
  - File f = new File("directory/file.txt");
  - String fn = f.getAbsolutePath();
  - Name holds between programs
- Persistent
  - Values maintained after program exits
- But not too useful by itself
  - f.canRead(); f.canWrite(); f.delete();

OutputStream class

- OutputStream o = new FileOutputStream(f);
- o.write(int b);
  - Write the byte represented by b
    - Ignore parts >255
- o.write(byte[] b);
  - Write the array of bytes (in order)
- o.flush();
  - Make sure everything is persistent
- o.close();
  - Done

InputStream class

- InputStream i = new FileInputStream(f);
- int b = i.read();
  - Read one byte and place in b
- int count = i.read(byte[] b);
  - read b.length() bytes into b
  - If not enough to read, count < b.length
- i.reset()
  - Go back to the beginning
- long skipped = i.skip(long n);
  - Skip ahead n bytes (if not enough, skipped<n)
- i.close();
  - Done

Scanner class

- Scanner s = new Scanner(f);
- String st = s.next();
  - Get the next space-terminated string
- int I = s.nextInt();
- float f = s.nextFloat();
- …
- PrintWriter class used to write values to be read with a Scanner

A stream is like:

A. An Array
   - Items in sequence starting from the beginning
   - Can't remove something from the middle

B. A Linked List
   A. Can't "jump around", must access in order

C. Both

D. Neither

PURDUE UNIVERSITY

CS18000: Problem Solving And Object-Oriented Programming

Buffered I/O, Network I/O

20 April 2011

Prof. Chris Clifton

COMPUTER SCIENCES

(Generic) Stream Operations

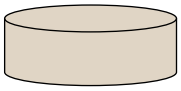
- open
  - Make ready for reading (at beginning), writing (at beginning or at end)
  - Generally done by a Stream *constructor* in Java
- read / write
  - Get data from stream or put data on stream
- skip
  - Move ahead (or back) some distance in stream
  - Equivalent to reading and throwing away what is read
- reset
  - Move to beginning of stream
- flush
  - Make sure data sent to stream
  - Used only when writing
- close
  - Make sure data sent to stream
  - Stream can't be used without "open"ing again
  - Others may use stream

I/O Latency Issues

- Devices normally read *blocks*
  - Typically 512 – 8192 bytes
  - Is your program ready for all that?
- Suppose you need one int (4 bytes)
  - Stream has device read the block to get it
- Then need the next int
  - Stream has device read the block again
- Fast disks read 140KB/millisecond
  - But up to 8 milliseconds to get back to the same block

Solution: Buffered I/O

- Stream reads an entire block (or more)
  - Saves it in memory
  - When you ask for the next byte, you get it
  - When you ask for a byte that isn't there, get the next block
- Dramatic performance improvements
  - But not always invisible to program



Buffered I/O

- Where is the buffering done?
  A. Do it yourself?
  B. Scanner object?
  C. FileInputStream object?
  D. Java Virtual Machine?
  E. Disk?
- Answer: Yes

Challenge: Multiple Buffers

- Okay as long as one reader or writer
  - Each goes to buffer at next layer
- But what if multiple readers?
  - Scanner s1 reads from file f
  - Scanner s2 reads from file f
- s1 will get more than it needs
  - s2 will "miss" data not yet "used" by s1

Issues with Buffered I/O

- A Scanner can check if the next value is a Float, Integer, etc. without reading it. To do this, it
  A. must use buffered I/O,
  B. probably uses buffered I/O, or it would be to slow to check then go back to get the next value, or
  C. Does not use buffered I/O, since what is in the buffer might change after it is checked.

Solution: Single object accesses each stream

- FileInputStream fis = new FileInputStream("abc.txt");
  - No other object should use file abc.txt
  - If another method needs to read abc.txt, *pass it fis*
    - Or make an instance variable of the class
- Use only single abstraction to access stream
  - If using a Scanner, don't access call methods of the InputStream once Scanner instantiated

Network I/O

- Network connection is a stream
  - One end writes data (OutputStream)
  - Other end reads data (InputStream)
  - *Some network connections have both*
- Can use higher-level abstractions
  - Scanner
  - Printwriter

Network I/O: Socket

- Endpoint of network stream
- Analogous to *File* class (but not quite)
  - FileInputStream constructed using a File
  - Socket has getInputStream(), getOutputStream()
- new Socket("www.cs.purdue.edu", 80);
  - Creates a (2-way) connection to given address

Network I/O: Use

- Constructing the socket opens an InputStream and OutputStream
- Read/Write using whatever tools you want
  - Read/Write bytes directly on stream
  - Build Scanner on InputStream, PrintWriter on OutputStream
- I/O is buffered
- Close when done
  - Closing streams separate from closing socket

Stream operations on a Network

- *reset* on a network stream
  A. should reload the web page
  B. should cause the next byte read to be the first byte sent
  C. should close the network connection
  D. doesn't make sense

Blocking I/O

- What if there is nothing to read?
  - File: This is an "end of file"
  - Network (or keyboard): the input may not be finished
- End-of-file only occurs when other end explicitly closed
- Blocking I/O: wait for input
  - Wait until enough bytes come in to fulfill read (e.g., a complete integer and terminator/space)
  - Then return
- *Also non-blocking I/O: Exception if nothing to read*

Network I/O: Server

- What if we don't know who will connect?
  - E.g., web server
- Solution: Listener
  - new ServerSocket(80); // Listen on port 80
- accept() method waits
  - returns a Socket when someone connects