CSSE 220 Day 27

Finish the Sorting Intro

Non-text Files, reading and Writing Objects

Work on Spellchecker Project

CSSE 220 Day 27

- Turn in written problem now.
- If you find a good dictionary to use, please post a link to it on the Mini-project discussion forum.
- Everything for the Mini-project is due at the beginning of your class time on Day 30. No late days may be used for this one.
    - Why?
    - Presentations in class that day
    - Graders are students, too.
- There will be time in class to work with your team every day. Do not miss it!

Written problems

- Due to end of term time pressures, I will no longer have due dates for written problems or collect them.
- I will assign problems and post solutions. When you get a chance, you should do them and check your answers.

Project presentation/demonstration

- Day 30 in class
- Informal and informational
- What does your program do? How does it do it
- Data Structures and algorithms.
- Intended audience: Your classmates
    - Already know what the project is.
    - Already know Java'
    - Already know the data structures involved.
- No more than 7 minutes, including Q&A time.
- Just before your presentation, we will randomly choose which of your team members will present, so everyone should be prepared to do it.

THE DEPARTMENT OF COMPUTER SCIENCE & SOFTWARE ENGINEERING

INVITES YOU TO THE

DIRECTOR OF SOFTWARE ENGINEERING FACULTY CANDIDATE TALK

SHAWN BOHNER
VIRGINIA TECH

SOFTWARE SYSTEMS CHANGE TOLERANCE: AN EVOLVING PERSPECTIVE

FRIDAY, FEBRUARY 8, 2008 4:30 PM
O269

Please stay afterward to talk informally with Shawn.

JP says that There will be pizza!

Second candidate talk

- Monday, 10th period
- John Georgas

- Title: Supporting Architecture- and Policy-based Self-Adaptive Software Systems

Today's Agenda

- Work on Spellchecker
- Finish the Sorting intro
- Begin random access files and serialization

Homework

- Substantial progress on SpellChecker
- Written problems – not to turn in.
- A few things to read from the internet.
- Document available later today.

Knowledge of Elementary Sorts

- What should you know/be able to do by the end of this course?
  - The basic idea of how each sort works
    - insertion, selection, bubble, shell, merge
  - Can write the code in a few minutes
    - insertion, bubble, selection
    - perhaps with a minor error or two
    - not because you memorized it, but because you understand it
  - What are the best case and worst case orderings of N data items? For each of these:
    - Number of comparisons
    - Number of data movements

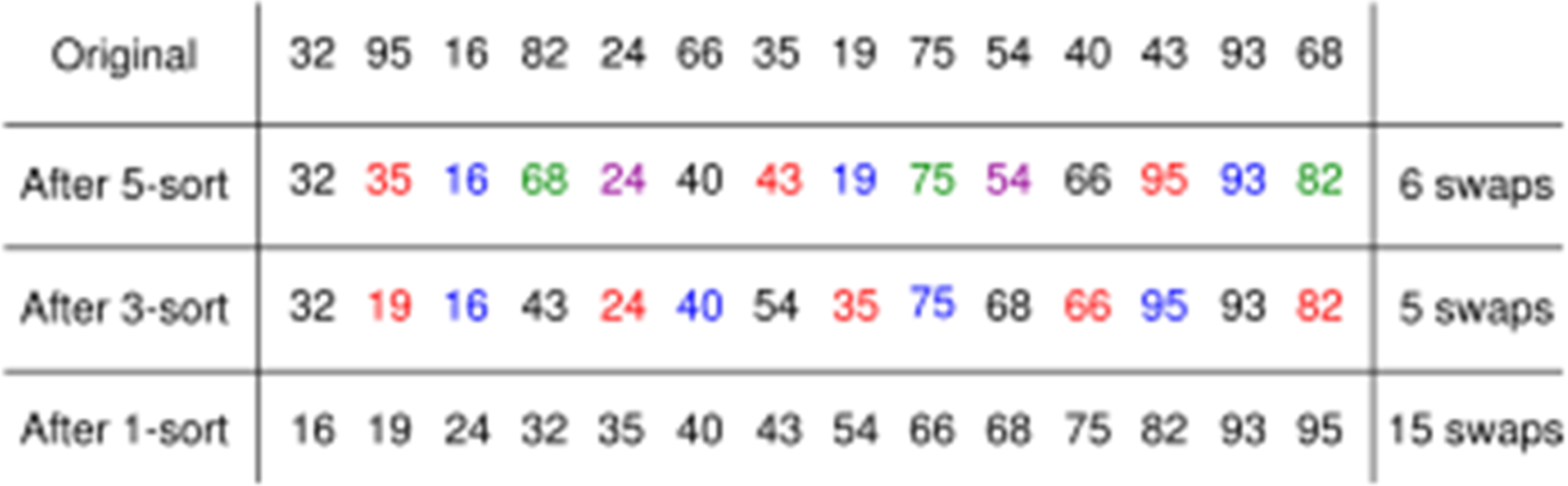
Elementary Sort summary

- Insertion sort
  - for (i=1; i< N; i++)
    - place a[i] in its correct position relative to a[0] …a[i-1]
      - move "right" each of those items that is less than a[i].
- Selection sort
  - for (i=N-1; i>0; i--)
    - maxPos = location of largest element among a[0] … a[i]
    - a[i]↔a[maxPos]
- Bubble sort
  - for (i=0; i< N-1; i++)
  - for (j=0; j≤ i; j++)
      - if (a[j] > a[j+1]) a[j]↔a[j+1]
- Demonstrations:
  - http://www.cs.ubc.ca/~harrison/Java/sorting-demo.html
  - http://www.geocities.com/siliconvalley/network/1854/Sort1.html

Shell sort

- 1959, Donald Shell
- Based on insertion sort
- Faster because it compares elements with a gap of several positions
- For example, if the gap size is 8,
  - Insertion sort elements 0, 8, 16, 24, 32, 40, …
  - Insertion sort elements 1, 9, 17, 25, 33, 41, …
  - …
  - Insertion sort elements 7, 15, 23, 31, 39, 47, …
- Elements that are far out of order are quickly moved closer to where they are supposed to go.

ShellSort example



Original	32	95	16	82	24	66	35	19	75	54	40	43	93	68	
After 5-sort	32	35	16	68	24	40	43	19	75	54	66	95	93	82	6 swaps
After 3-sort	32	19	16	43	24	40	54	35	75	68	66	95	93	82	5 swaps
After 1-sort	16	19	24	32	35	40	43	54	66	68	75	82	93	95	15 swaps

ShellSort Code

```
public static final int[] GAPS = {1, 4, 10, 23, 57, 132, 301, 701};

public static void shellSort(int[] a) {
  for (int gapIndex = GAPS.length - 1; gapIndex >= 0; gapIndex--) {
    int increment = GAPS[gapIndex];
    if (increment < a.length)
      for (int i = increment; i < a.length; i++) {
        int temp = a[i];
        for (int j = i;
             j >= increment && a[j - increment] > temp;
             j -= increment) {
             a[j] = a[j - increment];
           a[j - increment] = temp;
        }
      }
  }
}
```

TEST CODE:

```
public static void main(String[] args) {
      int SIZE = 31;
      int [] nums = new int[SIZE];
      for (int i=0; i<SIZE; i++) {
         nums[i] = (SIZE/2 + 5*i) % SIZE;
      }
      printArray("Before sort", nums);
      shellSort(nums);
      printArray("After sort", nums);
```

Shell sort gap sizes

- Start with a large gap
- Do it again with a smaller gap
- Keep decreasing the gap size
- The last time, the gap must be 1 (why?)
- No gap size should be a multiple of another (except all are multiples of 1)
- If proper gaps are chosen, worst-case performance is $O(N (\log N)^2)$
- An example of shellsort analysis (not for the faint of heart):
  - http://www.cs.princeton.edu/~rs/shell/paperF.pdf

Shellsort animation

- http://www.cs.princeton.edu/~rs/shell/animate.html

Merge Sort

- Divide and conquer
- Sort each half, merge halves together
- How to sort each half?
  - Use Merge sort
- Running time to merge two sorted arrays whose total length is N:
  - O(N)

```
public static void mergeSort( int [ ] a )
  {
      int [ ] tmpArray = new int[ a.length ];
      mergeSort( a, tmpArray, 0, a.length - 1 );
  }

  /**
   * Internal method that makes recursive calls.
   * @param a an array of Comparable items.
   * @param tmpArray an array to place the merged result.
   * @param left the left-most index of the subarray.
   * @param right the right-most index of the subarray.
   */
  private static void mergeSort( int [ ] a, int [ ] tmpArray,
             int left, int right )
  {
      if( left < right )
      {
          int center = ( left + right ) / 2;
          mergeSort( a, tmpArray, left, center );
          mergeSort( a, tmpArray, center + 1, right );
          merge( a, tmpArray, left, center + 1, right );
      }
  }
```

```
/**
   * Internal method that merges two sorted halves of a subarray.
   * @param a an array of Comparable items.
   * @param tmpArray an array to place the merged result.
   * @param leftPos the left-most index of the subarray.
   * @param rightPos the index of the start of the second half.
   * @param rightEnd the right-most index of the subarray.
   */
  private static void merge( int [ ] a, int [ ] tmpArray,
                             int leftPos, int rightPos, int rightEnd ) {
      int leftEnd = rightPos - 1;
      int tmpPos = leftPos;
      int numElements = rightEnd - leftPos + 1;

      // Main loop
      while( leftPos <= leftEnd && rightPos <= rightEnd )
          if( a[ leftPos ] <=  a[ rightPos ] )
              tmpArray[ tmpPos++ ] = a[ leftPos++ ];
          else
              tmpArray[ tmpPos++ ] = a[ rightPos++ ];

      while( leftPos <= leftEnd )    // Copy rest of first half
          tmpArray[ tmpPos++ ] = a[ leftPos++ ];

      while( rightPos <= rightEnd )  // Copy rest of right half
          tmpArray[ tmpPos++ ] = a[ rightPos++ ];

      // Copy tmpArray back
      for( int i = 0; i < numElements; i++, rightEnd-- )
          a[ rightEnd ] = tmpArray[ rightEnd ];
  }
```

Informal Analysis of Mergesort

- For simplicity, assume that N is a power of 2.
- N = Time for merging the sorted halves
- N = (N/2)*2 = time for merging four sorted "quarters" into two sorted "halves"
- N = (N/4)*4 = time for merging four *so*rted "eighths" into two sorted "quarters"
- …
- N = (2)*N/2 = time for merging N single elements into N/2 sorted pairs
- Total =

Java I/O (Input and Output) 1

- Back In the Day [TM]
  - I/O only involved a few possible sources/destinations
  - terminal, printer, card reader, hard disk
  - Typically there were separate sets of functions for each type of source or destination.
- Now there are many more sources/destinations
  - including network locations.
  - and we recognize that most of the I/O functions are common to all sources/destinations
- In order to make **all** I/O more flexible and adaptable in Java, **simple** I/O is more complex than in some other languages.

Java I/O (Input and Output) 2

- What is a Stream?
  - An abstract representation of information flow that is independent of the source and/or destination.
- A stream is One-Way
  - Either an Input Stream or an Output Stream
- InputStream
  - Subclasses include FileInputStream, ObjectInputStream, AudioInputStream.
  - A socket has a **getInputStream** method that lets us get info from a network connection.
  - **System.in** is an InputStream
- OutputStream
  - Subclasses include FileOutputStream, ObjectOutputStream.
  - A **PrintStream** is a specialized OutputStream with characteristics suitable for standard output.
  - **System.out** is a PrintStream.

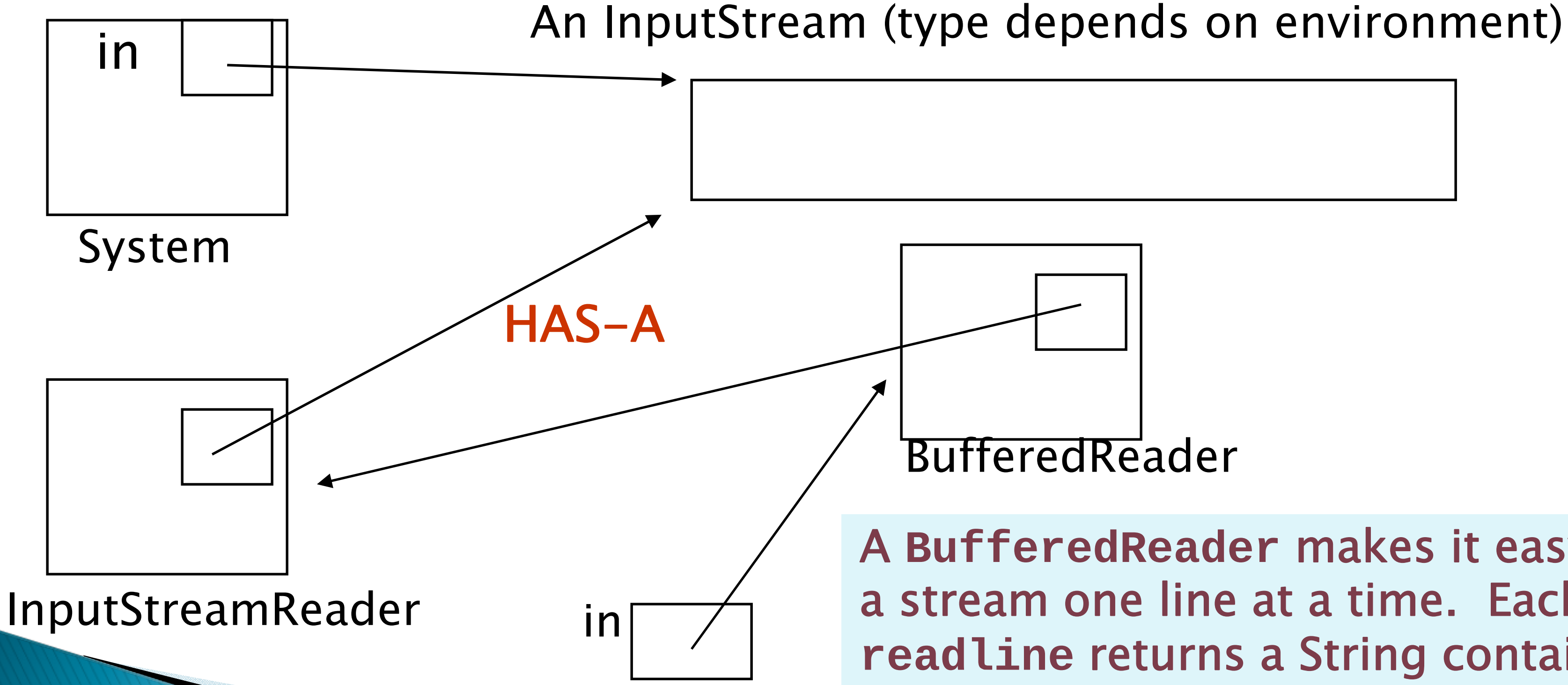
Java I/O (Input and Output) 3

- Three pre-defined streams
  - System.in (an InputStream)
  - System.out (a PrintStream)
  - System.err (a PrintStream)
- Streams are byte-oriented.
  They read or write bytes or arrays of bytes.
- Readers and Writers are character-oriented, they read or write characters or arrays of characters.
- Examples of Reader classes:
  - InputStreamReader, BufferedReader, FileReader, PushBackReader, StringReader.
- Examples of Writer classes:
  - OutputStreamWriter, PrintWriter, BufferedWriter, StringWriter

Reader Construction – From `System.in`

Line-at-a-time input from the standard input stream **`System.in`**

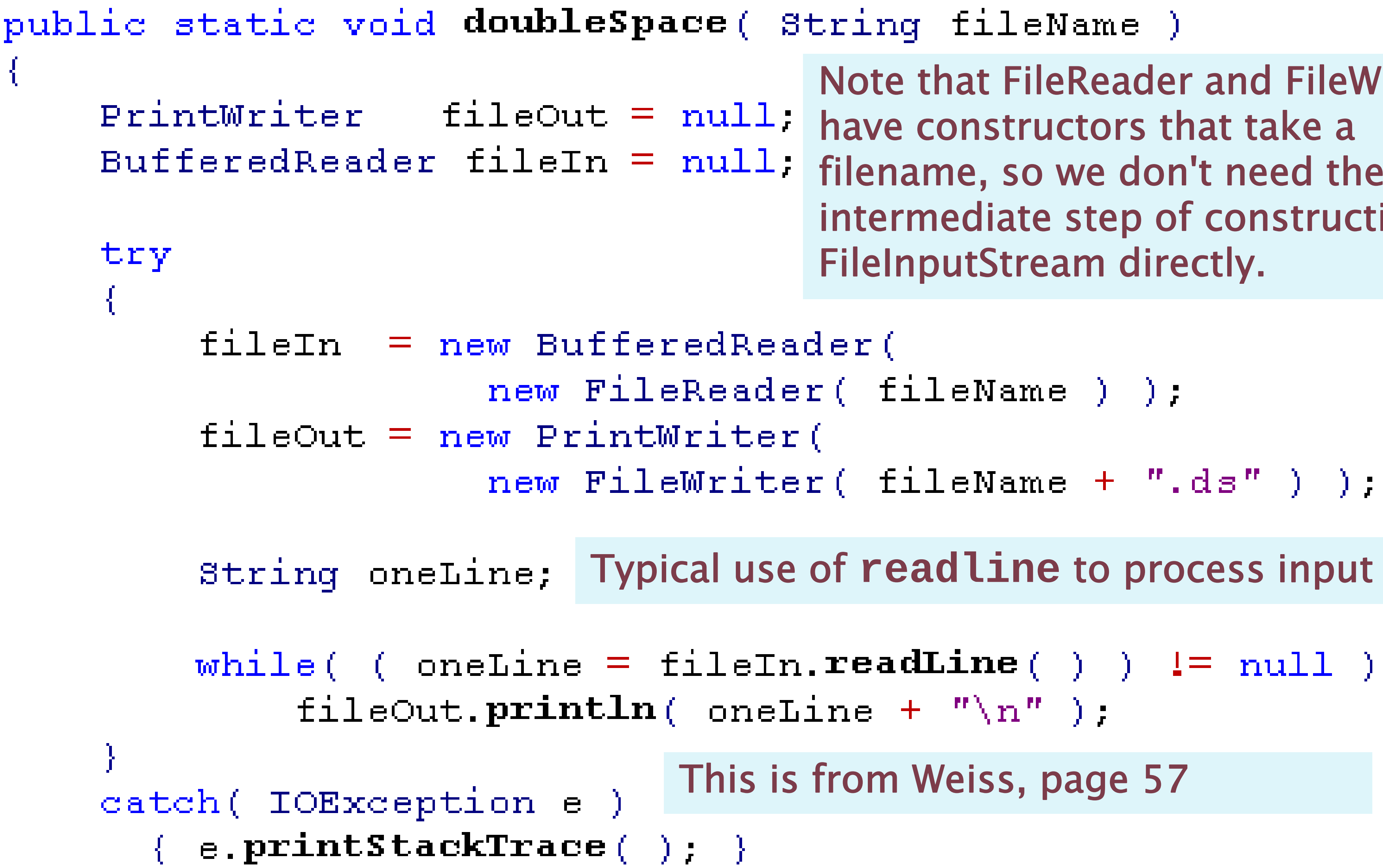
```
BufferedReader in = new BufferedReader(
                         new InputStreamReader( System.in ) );
```



A **`BufferedReader`** makes it easy to read a stream one line at a time. Each call to **`readline`** returns a String containing the next input line (without the end-of-line character).

Reader/Writer Construction – From files

I/O to/from files using a BufferedReader and a PrintWriter.



```java
public static void doubleSpace( String fileName )
{
    PrintWriter    fileOut = null;
    BufferedReader fileIn = null;

    try
    {
        fileIn  = new BufferedReader(
                      new FileReader( fileName ) );
        fileOut = new PrintWriter(
                      new FileWriter( fileName + ".ds" ) );

        String oneLine;

        while( ( oneLine = fileIn.readLine( ) ) != null )
            fileOut.println( oneLine + "\n" );
    }
    catch( IOException e )
      { e.printStackTrace( ); }
```

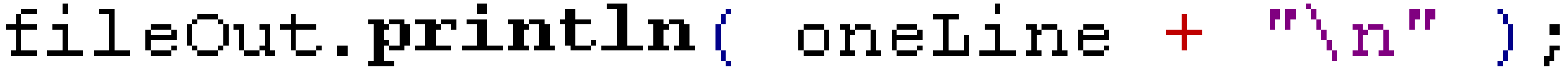
Note that FileReader and FileWriter have constructors that take a filename, so we don't need the intermediate step of constructing an FileInputStream directly.

Typical use of **readline** to process input

This is from Weiss, page 57

Weiss's one bad idea in that example

Can you see what is not so good about the code on the previous slide?



```
fileOut.println( oneLine + "\n" );
```

What should we do instead?

```
System.getProperty("line.separator");
```