CS 121 - Intro to Java - Lecture 15

Announcements
• Diffy mini-project due 3/12
• Program 5 up, due 3/22
• Ch 8 embedded due 3/22 (trapdoor req)
• Ch 7 OWL due 3/23
• Final 5/11 (second midterm online 2nd wk Apr)
• Midterm grades should be final by noon today
• Drop dead date: 3/23
• How to drop – get a signature at CS BLDG room 100 (main office)
Topics for today:
Some more ambitious programs

Inverse Manhattan Problem
Word Frequency
Eratosthenes- Sieve for primes
The Manhattan Island problem

According to legend, the Indians who were living on Manhattan Island in 1626 sold the island to Dutch settlers for $24 worth of junk jewelry.

However, compound interest being what it is, $24 invested at - say 5% for 380 years - will generate a substantial sum of money.

The Inverse Manhattan problem: If the Island today is worth - let's say - one trillion dollars, at what rate should the aboriginal peoples have invested so that they could buy back the island today?
We’ll write a CompoundInterest class that we can use to solve this and related problems

compounded 1 time/year:

$$A(t) = A_0 (1+r)^t$$

compounded k times/year:

$$A(t) = A_0 (1+r/k)^{tk}$$
public class CompoundInterest{

    private double rate;
    private double amount;
    private double fee = 0.0; // yearly fee
    private int years;
    final double tolerance = 100.00; // for findRate

    public double getAmount(){ return amount; }
    public double getFee(){ return fee; }
    public double getRate(){return rate; }

    public void setFee(double fee){this.fee = fee; }
}
public CompoundInterest(double r, double start, int years){
    amount = start;  rate = r;
    this.years = years;
}

public CompoundInterest(double r, double start, int years, double fee){
    amount = start;
    rate = r;
    this.fee = fee;
    this.years = years;
}

public CompoundInterest(){
    amount = 0.0;
    rate = 0.0;
    years = 0;
public double calcAmt()
{
    double amt = amount;
    for (int y = 0; y < years; y++)
        amt = (1 + rate)*amt - fee;
    return amt;
}

public double calcAmt(double someAmt, int ys, double r)
{
    double amt = someAmt;
    for (int y = 0; y < ys; y++)
        amt = (1 + r)*amt - fee;
    return amt;
}
Notice:
Multiple constructors
Multiple versions of the same method
Java tells them apart by *signature*
The inverse Manhattan problem: Given a start amount, a target amount, a number of years - 

What interest rate gets you from start to target?

For example: $10K -> $28K, 9 years
First panel: spread of 10% between mid and high
Second panel: spread of 5% between low and mid
Next try: 12.5%
public double findRate(double amt, double target, int years) {
    double low = 0.0;
    double high = findUpBound(amt, target, years);
    double mid = (high + low)/2; // mid: middle rate
    double curRate = mid;
    double curAmt = calcAmt(amt, years, mid);
    while (Math.abs(curAmt - target) > tolerance) {
        System.out.println("current middle rate: "+ mid);
        if (curAmt > target) {
            high = mid;
            mid = (mid + low)/2; }
        else {
            low = mid;
            mid = (mid + high)/2; }
        curAmt = calcAmt(amt, years, mid);
    } // end loop
    return mid;
}
Notice: an important helper method -

findUpBound

How does it do what it does?

What role does it play in the method it serves?
public double findUpBound
    (double amt, double target, int years){
    final double increment = .10;
    // guess, up by 10% each year
    double rate = increment;
    while(calcAmt(amt,years,rate) < target){
        rate = rate + increment;
    }
    return rate;
}
// note how the the while loop is used!
This method - binary search
A dominant idea in computer science.

Example: spell check
import java.util.Scanner;

public class CompoundTester2 {

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter a starting amount");
        double amt = scan.nextDouble();
        System.out.println("Enter target");
        double target = scan.nextDouble();
        System.out.println("Enter number of years");
        int years = scan.nextInt();
        CompoundInterest c = new CompoundInterest();
        double rate = c.findRate(amt, target, years);
        System.out.printf("rate required %6.4f \n", rate);
    }
}
As text indicates:
$24
380 years
Target of one trillion dollars:
rate required: 0.0665 (6.65%)
What is the distribution of words in a text?

For example: the Federalist Papers, #49 - written by Alexander Hamilton, or James Madison?

It was known for certain that H wrote some, M wrote others.

Statistical analysis (word frequencies, mostly) of the known materials were sufficient to identify the author of the papers in dispute (including #49) - this, in the 1950's
The author of the "Notes on the State of Virginia," quoted in the last paper, has subjoined to that valuable work the draught of a constitution, which had been prepared in order to be laid before a convention, expected to be called in 1783, by the legislature, for the establishment of a constitution for that commonwealth. The plan, like every thing from the same pen, marks a turn of thinking, original, comprehensive, and accurate; and is the more worthy of attention as it equally displays a fervent attachment to republican government and an enlightened view of the dangerous propensities against which it ought to be guarded. One of the precautions which he proposes, and on which he appears ultimately to rely as a palladium to the weaker departments of power against the invasions of the stronger, is perhaps altogether his own, and as it immediately relates to the subject of our present inquiry, ought not to be overlooked.

His proposition is, "that whenever any two of the three branches of government shall concur in opinion, each by the voices of two thirds of their whole number, that a convention is necessary for altering the constitution, or CORRECTING BREACHES OF IT, a convention shall be called for the purpose."
A simple word frequency program
Reads in multiple lines of text (end with empty line: 2<CRLF>)

Break the lines into words (called tokens)

Store words in an array, keeping track of word multiplicities

Five participating classes - most complex program we've seen so far...
StringTokenizer heads up..

Turns

“now is the time for all, all”

Into

now
is
the
time
for
all, [what about that pesky comma??]
all
<table>
<thead>
<tr>
<th>the</th>
<th>fox</th>
<th>......................</th>
<th>good</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

WordData objects in array in WordStore
Design -

• Get the data (driver)
• Process the data (driver, WordStore class)
• Report results (done in driver, work in WordStore)
import java.util.*;
public class WordDriver{
    public static void main(String[] args){
        WordStore store = new WordStore();
        Scanner scan = new Scanner(System.in);
        StringTokenizer str;
        String t = " "; String s;
        System.out.println("Enter lines, two returns to end");
        while(t.length() > 0){
            t = scan.nextLine(); // read next line
            t = t.toLowerCase(); // convert line to all lower case
            str = new StringTokenizer(t,\"\t\n\r\f,?.!;:\");
            while(str.hasMoreTokens()){
                s = str.nextToken();
                store.updateWords(s); }
        }
        store.wordReport(); } }
An alternate design:

Get the line and hand *that* to WordStore object store.

Let store handle the job of pulling the line apart to get at the words...

How would this alter the driver?
How would this alter the constructor?
The WordStore class?
public class WordData{
    private String word;
    private int count;
    public WordData(String w){
        word = w;
        count = 1; }

    public String getWord(){ return word;}
    public int getCount(){return count;}

    public void incCount() {count++;}
    public String toString() {
        return(word + " ---- " + count); 
    }
}

public class WordStore{
    final int LAST_WORD = 100;
    private WordData[] words =
            new WordData[LAST_WORD];
    private int lastEmpty = 0;

    public void updateWords(String w){
        int where = -1; /*-1 means: word not yet found*/
        where = findWord(w);
        if (where >= 0) words[where].incCount();
        else addWord(w);
    }
}
public int findWord(String w){
    int ans = -1;
    for (int j=0; j < lastEmpty; j++){
        if (w.equals(words[j].getWord())){
            ans = j;
            break;
        }
    }
    return ans;
}

Why w.equals?? (why not ==)??
> String d1 = "dog";
> String d2 = "dog";
> d1 == d2
false   [cell model !!!]

BUT

> d1.equals(d2)
true

The sad truth - write your own equals
public void addWord(String w) {
    if (lastEmpty == LAST_WORD)
        System.out.println("store full " + w + " not added");
    else {
        words[lastEmpty] = new WordData(w);
        lastEmpty++;
    }
}

public void wordReport() {
    for (int j = 0; j < lastEmpty; j++)
        System.out.println(words[j].toString());
}
The StringTokenizer class (in java.util)

It takes a String, chops it up into pieces - tokens

We would like to look at

Now is the time, comrades!

And get back, in succession

Now.. is.. the.. time, .. comrades!

or

Now.. is.. the.. time .. comrades
StringTokenizer does this for us.

Two key constructors, Two key methods:

If str is a StringTokenizer (associated with a particular String), then these methods:

hasMoreTokens()
nextToken()

will do the job

while (str.hasMoreTokens()){
    System.out.println(str.nextToken());
}
import java.util.StringTokenizer;

public class TokenizerTest{
    public static void main(String[] args){
        StringTokenizer str;
        Scanner scan = new Scanner(System.in);
        System.out.println("enter a line of text");
        String s = scan.nextLine();
        str = new StringTokenizer(s);
        while (str.hasMoreTokens()){
            System.out.println(str.nextToken());
        }
    }
}

enter a line of text
now, yes now is the time!

now,
yes
now
is
the
time!
import java.util.*;
public class TokenizerTest{
    public static void main(String[] args){
        StringTokenizer str;
        Scanner scan = new Scanner(System.in);
        System.out.println("enter a line of text");
        String s = scan.nextLine();
        str = new StringTokenizer(s," ,.");
        while (str.hasMoreTokens()){
            System.out.println(str.nextToken());
        }
    }
}

The crucial line: notice 2nd argument
now, yes now is the time!

So StringTokenizer treats “,” as a sort of white space, but not “!”
So what’s a prime number?

2, 11, 13, 17, 19 - NO proper divisors

6, 15, 21, 51 (= 3x17!) are not primes

Old concept - Eratosthenes (275-195 BC) came up with a *Sieve* method for finding primes up to some number n.
\[ x^2 > 2 \]
\[ x^3 > 3 \]
\[ x^5 > 5 \]
Basic Idea of implementation:
An array of booleans.
“true” at cell j means: “ j is not a multiple of any earlier number”

Initially array is all true

First pass: all multiples of 2 (e.g. 4, 6, 8, ...) set to false
Then: all multiples of 3 (e.g. 6, 9, 12, ...) set to false (why 3: it’s first true after 2)

Then: 5 (skip 4: it’s already false-> mult of 2)
// we'll do primes < 100
public class SieveTester {
    public static void main(String[] args) {
        Sieve s = new Sieve(100);
        s.init(); // initialize row of numbers
        s.process(); // run sieve process
        s.report(); // report primes
    }
}
public class Sieve {
    int top;
    boolean[] nums;
    public Sieve(int cap) {
        top = cap;
    }
    public void init() {
        nums = new boolean[top];
        for (int j = 0; j < top; j++) nums[j] = true;
    }
    // Note- sieve (array) size decided at run-time
public void process()
{
    for(int j = 2; j < top; j++)
        if(nums[j] == true) killMultiples(j);
}

public void report()
{
    for(int j = 2; j < top; j++){
        if(nums[j]) System.out.println(j);
    }
}

public void killMultiples(int k){ // kills 2*k, 3*k etc
    for(int mult = 2 ; mult*k < top; mult++){
        nums[mult*k] = false; }
}