

CSC 222: Object-Oriented Programming

Fall 2017

Object-oriented design

- example: letter frequencies
ArrayLists vs. arrays, autoboxing
- example: word frequencies w/ parallel lists
exception handling, System.out.format
- example: word frequencies w/ objects
- object-oriented design issues
cohesion & coupling

1

Letter frequency

one common tool for identifying the author of an unknown work is letter frequency

- i.e., count how frequently each of the letters is used in the work
- analysis has shown that an author will tend to have a consistent pattern of letter usage

will need 26 counters, one for each letter

- traverse each word and add to the corresponding counter for each character
- having a separate variable for each counter is not feasible
- instead have an ArrayList of 26 counters
`this.counts.get(0)` is the counter for 'a'
`this.counts.get(1)` is the counter for 'b'
...
`this.counts.get(25)` is the counter for 'z'

letter frequencies from the
Gettysburg address

a:	93 (9.2%)
b:	12 (1.2%)
c:	28 (2.8%)
d:	49 (4.9%)
e:	150 (14.9%)
f:	21 (2.1%)
g:	23 (2.3%)
h:	65 (6.4%)
i:	59 (5.8%)
j:	0 (0.0%)
k:	2 (0.2%)
l:	39 (3.9%)
m:	14 (1.4%)
n:	71 (7.0%)
o:	81 (8.0%)
p:	15 (1.5%)
q:	1 (0.1%)
r:	70 (6.9%)
s:	36 (3.6%)
t:	109 (10.8%)
u:	15 (1.5%)
v:	20 (2.0%)
w:	26 (2.6%)
x:	0 (0.0%)
y:	10 (1.0%)
z:	0 (0.0%)

2

Letter frequency example

initially, have ArrayList of 26 zeros:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

after processing "Fourscore" :

0	0	1	0	1	1	0	0	0	0	0	0	0	0	2	0	0	2	1	0	1	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

after processing the entire Gettysburg address

93	12	28	49	150	21	23	65	59	0	2	39	14	71	81	15	1	70	36	109	15	20	0	0	10	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

3

Autoboxing & unboxing

natural assumption: will store the frequency counts in an ArrayList of ints

```
private ArrayList<int> counts; // SORRY, WON'T WORK!
```

- unfortunately, ArrayLists can only store object types (i.e., no primitives)
- fortunately, there exists a class named `Integer` that encapsulates an `int` value

```
private ArrayList<Integer> counts;
```

- the Java compiler will automatically
 - convert an `int` value into an `Integer` object when you want to store it in an `ArrayList` (called *autoboxing*)
 - convert an `Integer` value back into an `int` when need to apply an arithmetic operation on it (called *unboxing*)

BE CAREFUL: Java will not unbox an `Integer` for comparison

```
if (this.counts.get(0) == this.counts.get(1)) {  
    ...  
}
```

== will test to see if they are the same Integer objects

4

LetterFreq design

```
public class LetterFreq {
    private ArrayList<Integer> counts;
    private int numLetters;

    public LetterFreq(String fileName) throws java.io.FileNotFoundException {
        INITIALIZE this.counts AND this.numLetters

        FOR EACH WORD IN THE FILE
            FOR EACH CHARACTER IN THE WORD
                IF THE CHARACTER IS A LETTER
                    DETERMINE ITS POSITION IN THE ALPHABET
                    INCREMENT THE CORRESPONDING COUNT IN this.counts
                    INCREMENT this.numLetters
            }
    }

    public int getCount(char ch) {
        IF ch IS A LETTER
            DETERMINE ITS POSITION IN THE ALPHABET
            ACCESS & RETURN THE CORRESPONDING COUNT IN this.counts
        OTHERWISE
            RETURN 0
    }

    public double getPercentage(char ch) {
        IF ch IS A LETTER
            DETERMINE ITS POSITION IN THE ALPHABET
            ACCESS THE CORRESPONDING COUNT IN this.counts
            CALCULATE & RETURN THE PERCENTAGE
        OTHERWISE
            RETURN 0.0
    }

    public void showCounts() {
        FOR EACH LETTER IN THE ALPHABET
            DISPLAY THE LETTER, ITS COUNT & PERCENTAGE
    }
}
```

5

LetterFreq implementation

```
import java.util.ArrayList;
import java.util.Scanner;
import java.io.File;

public class LetterFreq {
    private final static String LETTERS = "abcdefghijklmnopqrstuvwxyz";
    private ArrayList<Integer> counts;
    private int numLetters;

    public LetterFreq(String fileName) throws java.io.FileNotFoundException {
        this.counts = new ArrayList<Integer>();
        for (int i = 0; i < LetterFreq.LETTERS.length(); i++) {
            this.counts.add(0);
        }
        this.numLetters = 0;

        Scanner infile = new Scanner(new File(fileName));
        while (infile.hasNext()) {
            String nextWord = infile.next();

            for (int c = 0; c < nextWord.length(); c++) {
                char ch = nextWord.charAt(c);
                int index = LetterFreq.LETTERS.indexOf(Character.toLowerCase(ch));
                if (index >= 0) {
                    this.counts.set(index, this.counts.get(index)+1);
                    this.numLetters++;
                }
            }
        }
    }
}
```

will use a constant to store the alphabet

initialize the letter counts

for each word, process each letter ...

... get letter's index, increment its count

6

LetterFreq implementation (cont.)

```

    ...

    public int getCount(char ch) {
        int index = LetterFreq.LETTERS.indexOf(Character.toLowerCase(ch));
        if (index >= 0) {
            return this.counts.get(index);
        }
        else {
            return 0;
        }
    }

    public double getPercentage(char ch) {
        int index = LetterFreq.LETTERS.indexOf(Character.toLowerCase(ch));
        if (index >= 0 && this.numLetters > 0) {
            double percent = 100.0*this.counts.get(index)/this.numLetters;
            return Math.round(10.0*percent)/10.0;
        }
        else {
            return 0.0;
        }
    }

    public void showCounts() {
        for (int i = 0; i < LetterFreq.LETTERS.length(); i++) {
            char ch = LetterFreq.LETTERS.charAt(i);
            System.out.println(ch + ": " + this.getCount(ch) + "\t(" +
                this.getPercentage(ch) + "%)");
        }
    }
}

```

if it is a letter,
access & return
its count

if it is a letter,
calculate & return
its percentage

display all letters,
counts and
percentages

7

Interesting comparisons

letter frequencies from the Gettysburg address	letter frequencies from Alice in Wonderland	letter frequencies from Theory of Relativity
a: 93 (9.2%)	a: 8791 (8.2%)	a: 10936 (7.6%)
b: 12 (1.2%)	b: 1475 (1.4%)	b: 1956 (1.4%)
c: 28 (2.8%)	c: 2398 (2.2%)	c: 5272 (3.7%)
d: 49 (4.9%)	d: 4930 (4.6%)	d: 4392 (3.1%)
e: 150 (14.9%)	e: 13572 (12.6%)	e: 18579 (12.9%)
f: 21 (2.1%)	f: 2000 (1.9%)	f: 4228 (2.9%)
g: 23 (2.3%)	g: 2531 (2.4%)	g: 2114 (1.5%)
h: 65 (6.4%)	h: 7373 (6.8%)	h: 7607 (5.3%)
i: 59 (5.8%)	i: 7510 (7.0%)	i: 11937 (8.3%)
j: 0 (0.0%)	j: 146 (0.1%)	j: 106 (0.1%)
k: 2 (0.2%)	k: 1158 (1.1%)	k: 568 (0.4%)
l: 39 (3.9%)	l: 4713 (4.4%)	l: 5697 (4.0%)
m: 14 (1.4%)	m: 2104 (2.0%)	m: 3253 (2.3%)
n: 71 (7.0%)	n: 7013 (6.5%)	n: 9983 (6.9%)
o: 81 (8.0%)	o: 8145 (7.6%)	o: 11181 (7.8%)
p: 15 (1.5%)	p: 1524 (1.4%)	p: 2678 (1.9%)
q: 1 (0.1%)	q: 209 (0.2%)	q: 344 (0.2%)
r: 70 (6.9%)	r: 5437 (5.0%)	r: 8337 (5.8%)
s: 36 (3.6%)	s: 6500 (6.0%)	s: 8982 (6.2%)
t: 109 (10.8%)	t: 10686 (9.9%)	t: 15042 (10.5%)
u: 15 (1.5%)	u: 3465 (3.2%)	u: 3394 (2.4%)
v: 20 (2.0%)	v: 846 (0.8%)	v: 1737 (1.2%)
w: 26 (2.6%)	w: 2675 (2.5%)	w: 2506 (1.7%)
x: 0 (0.0%)	x: 148 (0.1%)	x: 537 (0.4%)
y: 10 (1.0%)	y: 2262 (2.1%)	y: 2446 (1.7%)
z: 0 (0.0%)	z: 78 (0.1%)	z: 115 (0.1%)

8

ArrayLists and arrays

ArrayList enables storing a collection of objects under one name

- can easily access and update items using `get` and `set`
- can easily `add` and `remove` items, and shifting occurs automatically
- can pass the collection to a method as a single object

ArrayList is built on top of a more fundamental Java data structure:
the *array*

- an array is a *contiguous, homogeneous* collection of items, accessible via an index
- arrays are much less flexible than ArrayLists
 - ✓ *the size of an array is fixed at creation, so you can't add items indefinitely*
 - ✓ *when you add/remove from the middle, it is up to you to shift items*
 - ✓ *you have to manually keep track of how many items are stored*
- for fixed size lists, arrays can be simpler

9

Arrays

to declare an array, designate the type of value stored followed by `[]`

```
String[] words;                int[] counters;
```

to create an array, must use `new` (an array is an object)

- specify the type and size inside brackets following `new`

```
words = new String[100];       counters = new int[26];
```

- or, if you know what the initial contents of the array should be, use shorthand:

```
int[] years = {2001, 2002, 2003, 2004, 2005};
```

to access or assign an item in an array, use brackets with the desired index

- similar to the `get` and `set` methods of ArrayList

```
String str = word[0];          // note: index starts at 0
                                // (similar to ArrayLists)
for (int i = 0; i < 26, i++) {
    counters[i] = 0;
}
```

10

LetterFreqArr implementation

```
import java.util.Scanner;
import java.io.File;

public class LetterFreqArr {
    private final static String LETTERS = "abcdefghijklmnopqrstuvwxyz";
    private int[] counts;
    private int numLetters;

    public LetterFreqArr(String fileName) throws java.io.FileNotFoundException {
        this.counts = new int[LetterFreqArr.LETTERS.length()];
        for (int i = 0; i < LetterFreqArr.LETTERS.length(); i++) {
            this.counts[i] = 0;
        }
        this.numLetters = 0;

        Scanner infile = new Scanner(new File(fileName));
        while (infile.hasNext()) {
            String nextWord = infile.next();

            for (int c = 0; c < nextWord.length(); c++) {
                char ch = nextWord.charAt(c);
                int index = LetterFreqArr.LETTERS.indexOf(Character.toLowerCase(ch));
                if (index >= 0) {
                    this.counts[index]++;
                    this.numLetters++;
                }
            }
        }
    }
}
```

could instead
make the field an
array

initialize array to
desired size

access/assign an
entry using []

increment is
simpler (no need
to get then set)

11

LetterFreqArr implementation (cont.)

```
. . .

public int getCount(char ch) {
    int index = LetterFreqArr.LETTERS.indexOf(Character.toLowerCase(ch));
    if (index >= 0) {
        return this.counts[index];
    }
    else {
        return 0;
    }
}

public double getPercentage(char ch) {
    int index = LetterFreqArr.LETTERS.indexOf(Character.toLowerCase(ch));
    if (index >= 0 && this.numLetters > 0) {
        double percent = 100.0*this.counts[index]/this.numLetters;
        return Math.round(10.0*percent)/10.0;
    }
    else {
        return 0.0;
    }
}

public void showCounts() {
    for (int i = 0; i < LetterFreqArr.LETTERS.length(); i++) {
        char ch = LetterFreqArr.LETTERS.charAt(i);
        System.out.println(ch + ": " + this.getCount(ch) + "\t(" +
            this.getPercentage(ch) + "%)");
    }
}
```

other method
essentially the
same (array
access uses []
instead of the get
method for
ArrayLists)

12

Why arrays?

general rule: ArrayLists are better, more abstract – USE THEM!

- they provide the basic array structure with many useful methods provided for free

```
get, set, add, size, contains, indexOf, remove, ...
```

- plus, the size of an ArrayList automatically adjusts as you add/remove items

when *might* you want to use an array?

- if the size of the list will never change and you merely want to access/assign items, then the advantages of arrays may be sufficient to warrant their use
 - ✓ if the initial contents are known, they can be assigned when the array is created

```
String[] answers = { "yes", "no", "maybe" };
```

- ✓ the [] notation allows for both access and assignment (instead of get & set)

```
int[] counts = new int[11];  
...  
counts[die1.roll() + die2.roll()]++;
```

- ✓ you can store primitive types directly, so no autoboxing/unboxing

13

Another example: word frequencies

now consider an extension of letter frequencies: we want a list of words and their frequencies

- i.e., keep track of how many times each word appears, report that number

basic algorithm: similar to LetterFreq except must store words & counts

```
while (STRINGS REMAIN TO BE READ) {  
    word = NEXT_WORD_IN_FILE;  
    word = word.toLowerCase();  
  
    if (ALREADY_STORED_IN_LIST) {  
        INCREMENT_THE_COUNT_FOR_THAT_WORD;  
    }  
    else {  
        ADD_TO_LIST_WITH_COUNT_OF_1;  
    }  
}
```

14

Parallel lists

we could maintain two different lists: one for words and one for counts

- `count.get(i)` is the number of times `word.get(i)` appears
- known as *parallel lists* since elements in parallel indices are related

"fourscore"	"and"	"seven"	"years"	...
0	1	2	3	

1	5	1	1	...
0	1	2	3	

15

WordFreq1

need two different lists for words & counts

- for each new word, check if already stored
- if so, increment its count
- if not, add the word & 1
- if you add a word but forget to add a count ...

```
public class WordFreq1 {
    private ArrayList<String> words
    private ArrayList<Integer> counts;
    private int totalWords;

    public WordFreq1(String fileName)
        throws java.io.FileNotFoundException {
        this.words = new ArrayList<String>();
        this.counts = new ArrayList<Integer>();
        this.totalWords = 0;

        Scanner infile = new Scanner(new File(fileName));
        while (infile.hasNext()) {
            String nextWord = infile.next().toLowerCase();
            int index = this.words.indexOf(nextWord);
            if (index >= 0) {
                this.counts.set(index, this.counts.get(index)+1);
            }
            else {
                this.words.add(nextWord);
                this.counts.add(1);
            }
            this.totalWords++;
        }
    }
    . . .
}
```

16

WordFreq1

getCount and
getPercentage
must search
this.words to
find the desired
word

- if found, access the corresponding count
- if not, must avoid index-out-of-bounds error

```
...
public int getCount(String str) {
    int index = this.words.indexOf(str.toLowerCase());
    if (index >= 0) {
        return this.counts.get(index);
    }
    else {
        return 0;
    }
}

public double getPercentage(String str) {
    int index = this.words.indexOf(str.toLowerCase());
    if (index >= 0) {
        double percent =
            100.0*this.counts.get(index)/this.totalWords;
        return Math.round(10.0*percent)/10.0;
    }
    else {
        return 0.0;
    }
}

public void showCounts() {
    for (String nextWord : this.words) {
        System.out.println(nextWord + ": " +
            this.getCount(nextWord) + "\t(" +
            this.getPercentage(nextWord) + "%)");
    }
}
}
```

17

Exception handling

recall: Java forces code
to acknowledge
potential (common)
errors

- if an exception (error) E is possible, the method can declare "throws E"
- alternatively, can use try-catch to specify what will happen

```
try {
    // CODE TO TRY
}
catch (EXCEPTION e) {
    // CODE TO HANDLE
}
```

```
public class WordFreq1 {
    private ArrayList<String> words;
    private ArrayList<Integer> counts;
    private int totalWords;

    public WordFreq1(String fileName) {
        this.words = new ArrayList<String>();
        this.counts = new ArrayList<Integer>();
        this.totalWords = 0;

        try {
            Scanner infile = new Scanner(new File(fileName));
            while (infile.hasNext()) {
                String nextWord = infile.next().toLowerCase();
                int index = this.words.indexOf(nextWord);
                if (index >= 0) {
                    this.counts.set(index, this.counts.get(index)+1);
                }
                else {
                    this.words.add(nextWord);
                    this.counts.add(1);
                }
                this.totalWords++;
            }
        }
        catch (java.io.FileNotFoundException e) {
            System.out.println("FILE NOT FOUND: " + fileName);
        }
    }
    ...
}
```

result: if file not found, error message,
empty lists, & program continues

18

Rounding vs. formatting

as is, `getPercentage` rounds the percentage to 1 decimal place

- not ideal
- better to store the number at full precision, round the display as desired

`System.out.format` allows you to control the format of output

```
...  
  
public double getPercentage(String str) {  
    int index = this.words.indexOf(str.toLowerCase());  
    if (index >= 0) {  
        double percent =  
            100.0*this.counts.get(index)/this.totalWords;  
        return Math.round(10.0*percent)/10.0;  
    }  
    else {  
        return 0.0;  
    }  
}  
  
public void showCounts() {  
    for (String nextWord : this.words) {  
        System.out.println(nextWord + ": " +  
            this.getCount(nextWord) + "\t(" +  
                this.getPercentage(nextWord) + "%)");  
    }  
}
```

19

System.out.format

general form:

```
System.out.format(FORMAT_STRING, VALUES);
```

- the format string is a string that contains the message to be printed, with placeholders for the values

%s	String value	%d	decimal (integer) value
%n	newline	%f	float (real) value

- can add field widths to placeholders

%8s	displays String (right-justified) in field of 8 characters
%-8s	displays String (left-justified) in field of 8 characters
%.2f	display float (right-justified) with 2 digits to right of decimal place
%6.2f	displays float (right-justified) in field of 6 chars, 2 digits to right of decimal

20

System.out.format examples

```
String name1 = "Chris";      String name2 = "Pat";
double score1 = 200.0/3;     double score2 = 89.9;

System.out.format("%s scored %f", name1, score1);

System.out.format("%s scored %5.1f", name1, score1);

System.out.format("%s scored %3.0f", name2, score2)

System.out.format("%-8s scored %f5.1%n", name1, score1);
System.out.format("%-8s scored %f5.1%n", name2, score2);
```

note: `System.out.format` is identical to `System.out.printf`

21

WordFreq1

better solution:

- `getPercentage` returns the actual percentage
- `showCount` formats the percentage when displaying
- if we want the words left-justified but ending in a colon, must add the colon to the value
- display the percent sign using `%%`

```
...
public double getPercentage(String str) {
    int index = this.words.indexOf(str.toLowerCase());
    if (index >= 0) {
        return 100.0*this.counts.get(index)/this.totalWords;
    }
    else {
        return 0.0;
    }
}

public void showCounts() {
    for (String nextWord : this.words) {
        System.out.format("%-15s %5d (%5.1f%%)\n", nextWord+":",
            this.getCount(nextWord),
            this.getPercentage(nextWord));
    }
}
```

22

Alternatively...

BIG PROBLEM WITH PARALLEL LISTS:

- have to keep the indices straight
- suppose we wanted to print the words & counts in alphabetical order
have to sort the lists, keep corresponding values together

BETTER YET:

- encapsulate the data and behavior of a word into a class
- need to store a word and its frequency → two fields (String and int)
- need to access word and frequency fields → `getWord` & `getFrequency` methods
- need to increment a frequency if existing word is encountered → `increment` method

23

Word class

```
public class Word {
    private String wordStr;
    private int count;

    public Word(String newWord) {
        this.wordStr = newWord;
        this.count = 1;
    }

    public String getWord() {
        return this.wordStr;
    }

    public int getFrequency() {
        return this.count;
    }

    public void increment() {
        this.count++;
    }

    public String toString() {
        return this.getWord() + ": " + this.getFrequency();
    }
}
```

a `Word` object stores a word and a count of its frequency

constructor stores a word and an initial count of 1

`getWord` and `getFrequency` are accessor methods

`increment` adds one to the count field

`toString` specifies the value that will be displayed when you print the `Word` object

24

WordFreq2

requires only one list
of Word objects

- .contains is no longer (directly) applicable
- must define our own method for searching the list for a word

*note: Java does provide
Map classes that are
even more ideal for this
application*

```
public class WordFreq2 {
    private ArrayList<Word> words;
    private int totalWords;

    public WordFreq2(String fileName) {
        this.words = new ArrayList<Word>();
        this.totalWords = 0;
        try {
            Scanner infile = new Scanner(new File(fileName));
            while (infile.hasNext()) {
                String nextWord = infile.next().toLowerCase();
                int index = this.findWord(nextWord);
                if (index >= 0) {
                    this.words.get(index).increment();
                }
                else {
                    this.words.add(new Word(nextWord));
                }
                this.totalWords++;
            }
        }
        catch (java.io.FileNotFoundException e) {
            System.out.println("FILE NOT FOUND: " + fileName);
        }
    }

    ////////////////////////////////////////////////////

    private int findWord(String desiredWord) {
        for (int i = 0; i < this.words.size(); i++) {
            if (this.words.get(i).getWord().equals(desiredWord)) {
                return i;
            }
        }
        return -1;
    }
}
```

25

WordFreq2

```
. . .

public int getCount(String str) {
    int index = this.findWord(str.toLowerCase());
    if (index >= 0) {
        return this.words.get(index).getFrequency();
    }
    else {
        return 0;
    }
}

public double getPercentage(String str) {
    int index = this.findWord(str.toLowerCase());
    if (index >= 0) {
        return 100.0*this.words.get(index).getFrequency()/this.totalWords;
    }
    else {
        return 0.0;
    }
}

public void showCounts() {
    for (Word nextWord : this.words) {
        System.out.format("%-20s (%5.1f%%)\n", nextWord,
            this.getPercentage(nextWord.getWord()));
    }
}

. . .
```

26

Object-oriented design

our design principles so far:

- a **class** should model some entity, encapsulating all of its state and behaviors
- a **method** should implement one behavior of an object
- a **field** should store some value that is part of the state of the object (and which must persist between method calls)
- fields should be declared private to avoid direct tampering – provide public accessor methods if needed
- **local variables** should store temporary values that are needed by a method in order to complete its task (e.g., loop counter for traversing an ArrayList)
- avoid duplication of code – if possible, factor out common code into a separate (private) method and call with the appropriate parameters to specialize

27

Cohesion

cohesion describes how well a unit of code maps to an entity or behavior

in a highly cohesive system:

- each class maps to a single, well-defined entity – encapsulating all of its internal state and external behaviors
- each method of the class maps to a single, well-defined behavior

advantages of cohesion:

- highly cohesive code is easier to read
 - don't have to keep track of all the things a method does
 - if method name is descriptive, makes it easy to follow code
- highly cohesive code is easier to reuse
 - if class cleanly models an entity, can reuse in any application that needs it
 - if a method cleanly implements a behavior, can be called by other methods and even reused in other classes

28

Coupling

coupling describes the interconnectedness of classes

in a loosely coupled system:

- each class is largely independent and communicates with other classes via a small, well-defined interface

advantages of loose coupling:

- loosely coupled classes make changes simpler
 - can modify the implementation of one class without affecting other classes
 - only changes to the interface (e.g., adding/removing methods, changing the parameters) affect other classes
- loosely coupled classes make development easier
 - you don't have to know how a class works in order to use it
 - since fields/local variables are encapsulated within a class/method, their names cannot conflict with the development of other classes.methods

29

Previous examples

- shapes
- Pig game
- roulette game
- word frequency

30

In-class exercise

`ufo.txt` is a text file that contains entries for > 50,000 UFO sightings

- each line lists the date (YYYY/MM/DD), state & city of a sighting

```
2010/08/13 NE Omaha
```

we want to be able store, process and search the sightings data

- we could store each sighting as a single String
but then would have to repeatedly extract the date/state/city as needed
- we could store the date, state & city in parallel lists
but it is risky to store related info in separate data structures
- best option: define a `UFOsighting` class to represent a sighting
can store all three pieces of data in a single object, provide methods to access
can then have a single `ArrayList` of `UFOsighting` objects

31

UFOsighting class

implement the `UFOsighting` class to provide the following functionality

Class UFOsighting	
<code>java.lang.Object</code>	↳ <code>UFOsighting</code>
<pre>public class UFOsighting extends java.lang.Object</pre>	
Record that contains information about a UFO sighting	
Version:	3/8/17
Author:	Dave Reed
Constructor Summary	
<code>UFOsighting</code> (<code>java.lang.String</code> dateString, <code>java.lang.String</code> stateString, <code>java.lang.String</code> cityString)	Constructs a UFO sighting object.
Method Summary	
<code>java.lang.String</code> <code>getCity</code> ()	Accessor for the sighting city.
<code>java.lang.String</code> <code>getDate</code> ()	Accessor for the sighting date.
<code>java.lang.String</code> <code>getState</code> ()	Accessor for the sighting state.
<code>java.lang.String</code> <code>toString</code> ()	Converts the sighting to a String.

32

UFOlookup class

a class for reading in and storing UFOsightings is provided

```
public class UFOlookup {
    private ArrayList<UFOsighting> sightings;

    public UFOlookup(String filename) {
        this.sightings = new ArrayList<UFOsighting>();

        try {
            Scanner infile = new Scanner(new File(filename));

            while (infile.hasNextLine()) {
                String date = infile.next();
                String state = infile.next();
                String city = infile.nextLine().trim();
                UFOsighting sight = new UFOsighting(date, state, city);
                this.sightings.add(sight);
            }
            infile.close();
        }
        catch (java.io.FileNotFoundException e) {
            System.out.println("No such file: " + filename);
        }
    }

    public int numSightings() {
        return this.sightings.size();
    }
}
```

download UFOlookup.java &
ufo.txt, then TRY IT OUT

33

Additions to UFOlookup

```
/**
 * Displays all of the sightings (one per line) that occurred in the specified
 * state, along with a final count of how many sighting there were.
 * @param state the state of interest (e.g., "NE")
 */
public void showByState(String state)
```

```
1943/06/01 NE Nebraska
1953/01/01 NE Nebraska (rural)
.
.
.
2010/07/16 NE Omaha
2010/08/04 NE Palisade
2010/08/13 NE Omaha
# of sightings = 317
```

```
/**
 * Displays all of the sightings (one per line) that occurred between the
 * specified dates, with a final count of how many sighting there were.
 * @param startDate the starting date (e.g., "1963/05/01")
 * @param endDate the ending date (e.g., "1963/05/31")
 */
public void showByDates(String startDate, String endDate)
```

```
1963/05/07 MA Lynn
1963/05/15 MD Towson
1963/05/15 NY New York City (Brooklyn)
# of sightings = 3
```

34