CSC 222: Object-Oriented Programming

Fall 2017

Object-oriented design

- example: word frequencies w/ parallel lists
- exception handling
- System.out.format
- example: word frequencies w/ objects
- object-oriented design issues
  cohesion & coupling

1

Another example: word frequencies

recall the LetterFreq application

- read in words from a file, count the number of occurrences of each letter

now consider an extension: 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

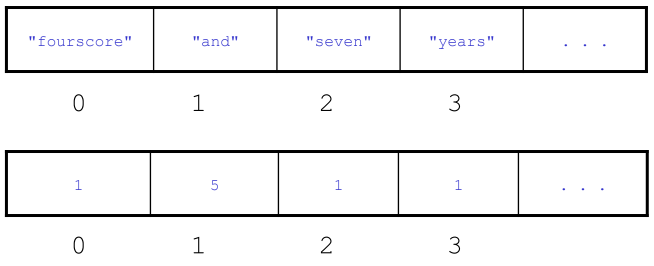
```
while (STRINGS_REMAIN_TO_BE_READ) {
    word = NEXT_WORD_IN_FILE;
    word = word.toLowercase();

    if (ALREADY_STORED_IN_LIST) {
        INCREMENT_THE_COUNT_FOR_THAT_WORD;
    }
    else {
        ADD_TO_LIST_WITH_COUNT_OF_1;
    }
}
```

2

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



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 = 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++;
    }
  }

  . . .
```

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) + "%)");
    }
  }
}
```

5

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 = 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

6

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) + "%)");
   }
 }
}
```

7

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

8

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`

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));
    }
  }
}
```

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

11

Word class

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

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

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

    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

12

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;
  }
}
```

13

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()));
  }
}

. . .
```

14

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

15

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

16

Coupling

coupling describes the interconnectedness of classes

in a loosely coupled system:

- each class is largely independent and communicates with other classes vi 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

17

Previous examples

- shapes
- skip-3 solitaire
- dot race
- roulette game
- word frequency

18

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

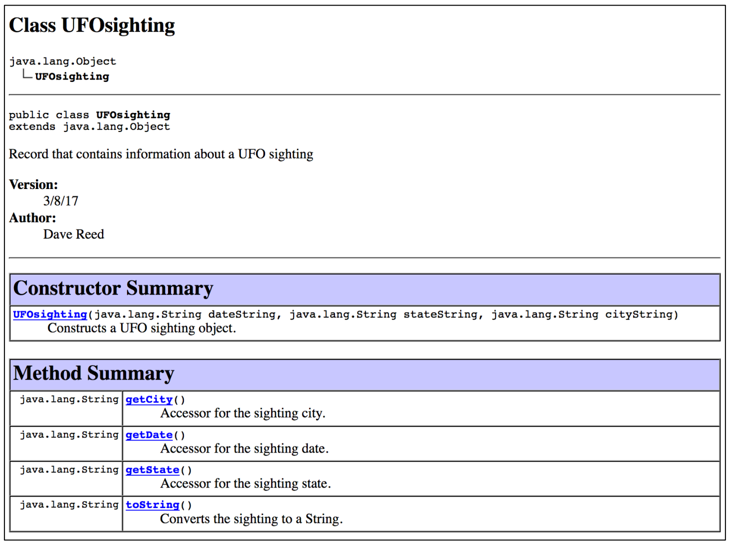
- we could store the date, state & city in parallel lists
  but it is risky to store related info in separate data structures

- 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

19

UFOsighting class

implement the `UFOsighting` class to provide the following functionality



Class UFOsighting

```
java.lang.Object
  └ UFOsighting
```

```
public class UFOsighting
extends java.lang.Object
```

Record that contains information about a UFO sighting

Version:
3/8/17
Author:
Dave Reed

Constructor Summary
`UFOsighting(java.lang.String dateString, java.lang.String stateString, java.lang.String cityString)` Constructs a UFO sighting object.

Method Summary	
`java.lang.String`	`getCity()` Accessor for the sighting city.
`java.lang.String`	`getDate()` Accessor for the sighting date.
`java.lang.String`	`getState()` Accessor for the sighting state.
`java.lang.String`	`toString()` Converts the sighting to a String.

20

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

21

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
```

22