CSC 321: Data Structures

Fall 2013

HW1 autopsy

1. parallel lists
2. Comparable Word objects
3. Comparators
4. MVC pattern

Approach 1: parallel lists

store words and counts in separate lists

"it"	"was"	"the"	"best"	…

2	5	12	1	…
0	1	2	3	…

- advantages?
- disadvantages?

while this approach is simple, it is not *cohesive*
- logically related data is not physically related
- it is up to you to keep words & counts in corresponding positions – tricky!

Issue: how do you sort the parallel lists?

can't use Collections.sort (at least not directly)

can modify the source code of a sort (e.g., selectionSort)

- whenever items in one list are moved, move corresponding items in the other
- could have two different versions (one to sort by frequency, one by alphabet), or could have a single, generic sort

```
public static <T1 extends Comparable<? super T1>, T2>  void
selectionSort(ArrayList<T1> items, ArrayList<T2> parallel) {
       for (int i = 0; i < items.size()-1; i++) {
         int indexOfMin = i;
         for (int j = i+1; j < items.size(); j++) {
           if (items.get(j).compareTo(items.get(indexOfMin)) < 0) {
             indexOfMin = j;
           }
         }

         T1 temp = items.get(i);
         items.set(i, items.get(indexOfMin));
         items.set(indexOfMin, temp);

         T2 tempP = parallel.get(i);
         parallel.set(i, parallel.get(indexOfMin));
         parallel.set(indexOfMin, tempP);
       }
   }
```

3

Approach 2: encapsulation

encapsulate a word and its frequency into a single object

- provide methods for accessing the fields, incrementing the frequency
- can make the Word class Comparable, so that a list of Words can be sorted

but we need two versions!

- compare by freq (for WordFrequencies)
- compare by alpha (for TagCloud)

```
public class Word implements Comparable<Word> {
    private String word;
    private int count;

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

    public Word(String newWord) {
        this(newWord, 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();
    }

    public int compareTo(Word other) {
        ???
    }
}
```

4

Issue: having two different Word classes is kludgy!

```
public class WordByFreq implements Comparable<WordByFreq> {
    private String word;
    private int count;

    . . .

    public int compareTo(WordByFreq other) {
        if (other.count == this.count) {
            return this.word.compareTo(other.word);
        } else {
            return (other.count - this.count);
        }
    }
}
```

```
public class WordByAlpha implements Comparable<WordByAlpha>
{
    private String word;
    private int count;

    . . .

    public int compareTo(WordByAlpha other) {
        if (other.word.equals(this.word)) {
            return (other.count - this.count);
        } else {
            return this.word.compareTo(other.word);
        }
    }
}
```

5

Approach 3: Comparators

the `Comparator` interface requires `compareTo` and `equals` methods

- can have one Word class, which provides 2 different Comparators
  (could be public constants, since they don't change)
  (or, could provide methods that return Comparators)

- there is a version of Collections.sort that takes a Comparator
  the specified Comparator determines the ordering when sorting

```
Collections.sort(words, Word.FreqComparator);
                              // will sort by frequency

Collections.sort(words, Word.AlphaComparator);
                              // will sort alphabetically
```

6

Issue

still ugly, but better than two separate classes

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

    . . .

    public static Comparator<Word> AlphaComparator = new Comparator<Word>() {
        public int compare(Word w1, Word w2) {
            if (w2.word.equals(w1.word)) {
                return (w2.count - w1.count);
            } else {
                return w1.word.compareTo(w2.word);
            }
        }

        public boolean equals(Word w1, Word w2) {
            return this.equals(w1, w2);
        }
    };

    public static Comparator<Word> FreqComparator = new Comparator<Word>() {
        public int compare(Word w1, Word w2) {
            if (w2.count == w1.count) {
                return w2.word.compareTo(w1.word);
            } else {
                return (w2.count - w1.count);
            }
        }

        public boolean equals(Word w1, Word w2) {
            return this.equals(w1, w2);
        }
    };
}
```

7

Approach 4: MVC

Model-View-Controller is a design pattern for OO software

- *Model* is the data & logic of the application
- *View* is the interface for viewing/updating the model
- *Controller* is the connector between the Model & View

ideally, there should be no I/O in the Model portion of the project

ideally, there should be (almost) no application logic in the Controller

GOAL: you should be able to change the View (e.g., go from console-based to GUI-based) without touching the Model

- simply change the Controller (which connects the new View with the old Model)

8

MVC version

the Word class encapsulates a word & its frequency

- with operations for storing a frequency, incrementing it, and comparing Words

the WordFreqList class encapsulates a list of Words

- with operations for adding words, getting frequencies, and retrieving the list in different orders

the Controller classes (WordFrequencies & TagCloud) then focus on I/O and calling Model methods

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

    public Word(String newWord, int count) { . . . }
    public Word(String newWord) { . . . }

    public String getWord() { . . . }
    public int getFrequency() { . . . }
    public void increment() { . . . }
    public String toString() { . . . }

    public static Comparator<Word> AlphaComparator =
        new Comparator<Word>() { . . . };
    public static Comparator<Word> FreqComparator =
        new Comparator<Word>() { . . . };
}
```

```
public class WordFreqList {
    private ArrayList<Word> words;
    private ArrayList<String> ignoreWords;
    private int maxSoFar = -1;

    public WordFreqList() { . . . }
    public WordFreqList(String ignoreFileName) { . . . }

    public void add(String newWord) { . . . }
    public void add(String newWord, int count) { . . . }
    public int getFrequency(String word) { . . . }
    public int getMax() { . . . }

    public List<Word> asListByFreq() { . . . }
    public List<Word> asListByAlpha() { . . . }
}
```

9

Other options?

how easily could these two programs be combined into one?

- i.e., go directly from speech.txt to speech.html

later in the course, you will learn about Maps

- a Map is a 2-dimensional structure that maps a key to a value
  e.g., can map a word to its frequency count
- makes it easy (and efficient) to look up and update a value based on its key
  e.g., look up the frequency count for a word

you will also learn about iterators

- instead of returning sorted copies of the Word list, you could provide iterators that allow for traversing the list

10