

CSC 221: Computer Programming I

Fall 2005

ArrayLists and structured data

- drawbacks of parallel lists
- parallel vs. structured lists
- example: word frequencies revisited
- type casting & exception handling
- example: card game

1

```
public class DocumentFreq
{
    private int numWords;
    private ArrayList<String> uniqueWords;
    private ArrayList<Integer> wordCounts;

    public DocumentFreq(String filename) throws java.io.FileNotFoundException {
        numWords = 0;
        uniqueWords = new ArrayList<String>();
        wordCounts = new ArrayList<Integer>();

        Scanner infile = new Scanner(new File(filename));
        while (infile.hasNext()) {
            String nextWord = infile.next();
            nextWord = nextWord.toLowerCase();
            nextWord = StringUtils.stripNonLetters(nextWord);

            numWords++;

            int index = uniqueWords.indexOf(nextWord);
            if (index >= 0) {
                int count = wordCounts.get(index);
                wordCounts.set(index, count+1);
            }
            else {
                uniqueWords.add(nextWord);
                wordCounts.add(1);
            }
        }
    }

    public int getNumWords() {
        return numWords;
    }

    public int getNumUniqueWords() {
        return uniqueWords.size();
    }

    public void showWords() {
        for (int i = 0; i < uniqueWords.size(); i++) {
            System.out.println(uniqueWords.get(i) + ": " + wordCounts.get(i));
        }
        System.out.println();
    }
}
```

Word frequencies revisited

recall the DocumentFreq class that processed a document and kept track of the number of words & unique words

it utilized parallel lists, containing the words and their associated counts

2

Drawbacks of parallel lists

while parallel lists are simple, they have several drawbacks

- since ArrayLists can only store objects, storing/manipulating the list of counts relies heavily on autoboxing/unboxing

recall: when you assign an `int` value to an `Integer` variable, the `int` is autoboxed

similarly: when you assign an `Integer` value to an `int` variable, or apply an `int` operator (`+`, `-`, `*`, `/`, `%`, `<`, `>`, `<=`, `>=`) to an `Integer` value, it is automatically unboxed

however: there are special cases that complicate things

```
if (wordCounts.get(i) == wordCounts.get(j)) {  
    . . .  
}
```

since `==` and `!=` are defined for objects, these `Integer`s are NOT unboxed – so, this does NOT perform the expected comparison

- more importantly, parallel lists are hard to maintain
 - ✓ it's easy to get the corresponding indices out of sync, then all is lost
 - ✓ if you ever want to store more information about each word (e.g., line numbers where it appears), then you have to add and maintain more parallel lists

3

Parallel vs. structured lists

a better approach when storing lists of related data is to structure the data

- we could define a class that encapsulates a word, including frequency (similar to the `Word` class we used in previous examples)

```
public class Word  
{  
    private String str;  
    private int freq;  
  
    public Word(String word)  
    {  
        str = word;  
        freq = 1;  
    }  
  
    public int getFrequency()  
    {  
        return freq;  
    }  
  
    public void incrementFrequency()  
    {  
        freq++;  
    }  
  
    . . .  
}
```

what other methods should we provide as part of `Word`?

4

Word class (cont.)

- to be general, we should provide a `toString` method for accessing the word
- in order to search an `ArrayList` of words, we also need to define `equals`
note: the `equals` method that is used to search a list (using `contains` or `indexOf`) takes an `Object` as parameter
`Object` is a generic class that encompasses all object types in Java

```
...  
public String toString()  
{  
    return str;  
}  
  
public boolean equals(Object other)  
{  
    try {  
        Word otherWord = (Word)other;  
        return str.equals(otherWord.toString());  
    }  
    catch (ClassCastException e) {  
        return false;  
    }  
}
```

can determine whether the `Object` passed to `equals` is another `Word` by attempting to cast the type, and catching an exception if that fails

in general:

```
try {  
    CODE WITH POTENTIAL FOR ERROR  
}  
catch (EXCEPTIONTYPE e) {  
    CODE FOR HANDLING THAT ERROR  
}
```

5

Word frequency using word

```
public class DocumentFreq  
{  
    private int numWords;  
    private ArrayList<Word> uniqueWords;  
  
    public DocumentFreq(String filename) throws java.io.FileNotFoundException {  
        numWords = 0;  
        uniqueWords = new ArrayList<Word>();  
  
        Scanner infile = new Scanner(new File(filename));  
        while (infile.hasNext()) {  
            String nextStr = infile.next();  
            nextStr = nextStr.toLowerCase();  
            nextStr = StringUtils.stripNonLetters(nextStr);  
  
            numWords++;  
  
            Word nextWord = new Word(nextStr);  
            int index = uniqueWords.indexOf(nextWord);  
            if (index >= 0) {  
                Word foundWord = uniqueWords.get(index);  
                foundWord.incrementFrequency();  
            }  
            else {  
                uniqueWords.add(nextWord);  
            }  
        }  
  
        public int getNumWords() {  
            return numWords;  
        }  
  
        public int getNumUniqueWords() {  
            return uniqueWords.size();  
        }  
  
        public void showWords() {  
            for (int i = 0; i < uniqueWords.size(); i++) {  
                System.out.println(uniqueWords.get(i) + ": " + uniqueWords.get(i).getFrequency());  
            }  
            System.out.println();  
        }  
    }  
}
```

now, we only need one list, with each list entry containing all the related information about a word

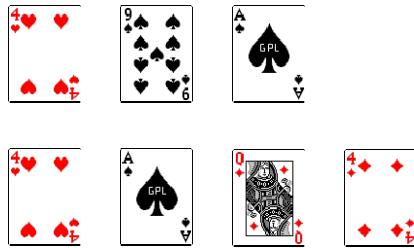
note that the search using `indexOf` works because `equals` is defined (and it ignores the frequency when comparing)

6

HW6 application

Skip-3 Solitaire:

- cards are dealt one at a time from a standard deck and placed in a single row
- if the rank or suit of a card matches the rank or suit either 1 or 3 cards to its left, then that card (and any cards beneath it) can be moved on top
- goal is to have the fewest piles when the deck is exhausted



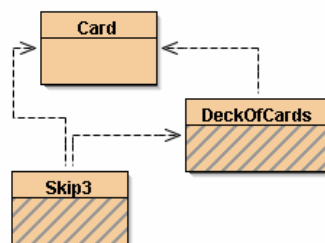
7

Skip-3 design

what are the entities involved in the game that must be modeled?

for each entity, what are its behaviors? what comprises its state?

which entity relies upon another? how do they interact?



8

Card & DeckOfCards

```
public class Card
{
    private char rank;
    private char suit;

    public Card(char cardRank, char cardSuit) { ... }

    public char getRank() { ... }
    public char getSuit() { ... }

    public String toString() { ... }
    public boolean equals(Object other) { ... }
}
```

card class encapsulates the behavior of a playing card

- cohesive?

DeckOfCards class encapsulates the behavior of a deck

- cohesive?
- coupling with card?

```
public class DeckOfCards
{
    private ArrayList<Card> deck;

    public DeckOfCards() { ... }

    public void shuffle() { ... }
    public Card dealCard() { ... }
    public void addCard(Card c) { ... }

    public int cardsRemaining() { ... }
    public String toString() { ... }
}
```

9

```
public class Card
{
    private char rank;
    private char suit;

    /**
     * Creates a card with the specified rank and suit
     * @param cardRank one of '2', '3', '4', ..., '9', 'T', 'J', 'Q', 'K', or 'A'
     * @param cardSuit one of 'S', 'H', 'D', 'C'
     */
    public Card(char cardRank, char cardSuit)
    {
        rank = cardRank;
        suit = cardSuit;
    }

    /**
     * @return the rank of the card (e.g., '5' or 'J')
     */
    public char getRank()
    {
        return rank;
    }

    /**
     * @return the suit of the card (e.g., 'S' or 'C')
     */
    public char getSuit()
    {
        return suit;
    }

    ...
}
```

Card class

10

Card class (cont.)

```
...  
  
/**  
 * @return a string consisting of rank followed by suit (e.g., "2H" or "QD")  
 */  
public String toString()  
{  
    return "" + rank + suit;  
}  
  
/**  
 * @return true if other is a Card with the same rank and suit, else false  
 */  
public boolean equals(Object other)  
{  
    try {  
        Card otherCard = (Card)other;  
        return (getRank() == otherCard.getRank() && getSuit() == otherCard.getSuit());  
    }  
    catch (ClassCastException e) {  
        return false;  
    }  
}
```

why is ""+ at the front of this expression?

again, when defining equals for a Card, we must attempt to cast the Object parameter to the type Card, and catch the exception if that fails

11

DeckOfCards class

```
public class DeckOfCards  
{  
    private ArrayList<Card> deck;  
  
    /**  
     * Creates a deck of 52 cards in order.  
     */  
    public DeckOfCards()  
    {  
        deck = new ArrayList<Card>();  
  
        String suits = "SHDC";  
        String ranks = "23456789TJQKA";  
        for (int s = 0; s < suits.length(); s++) {  
            for (int r = 0; r < ranks.length(); r++) {  
                Card c = new Card(ranks.charAt(r), suits.charAt(s));  
                deck.add(c);  
            }  
        }  
    }  
  
    /** @return the number of cards remaining in the deck  
     */  
    public int cardsRemaining()  
    {  
        return deck.size();  
    }  
  
    /** @return a String representation of the deck  
     */  
    public String toString()  
    {  
        return deck.toString();  
    }  
}
```

the constructor steps through each suit-rank pair, creates that card, and stores it in the ArrayList

the number of cards remaining is the current size of the ArrayList

toString is provided for debugging purposes

12

DeckOfCards class

```
...  
/**  
 * Shuffles the deck so that the locations of the cards are random.  
 */  
public void shuffle()  
{  
    for (int c = deck.size()-1; c > 0; c--) {  
        int swapIndex = (int)(Math.random()*(c+1));  
  
        Card tempCard = deck.get(c);  
        deck.set(c, deck.get(swapIndex));  
        deck.set(swapIndex, tempCard);  
    }  
}  
  
/**  
 * Deals a card from the top of the deck, removing it from the deck.  
 * @return the card that was at the top (i.e., end of the ArrayList)  
 */  
public Card dealCard()  
{  
    return deck.remove(deck.size()-1);  
}  
  
/**  
 * Adds a card to the bottom of the deck.  
 * @param c the card to be added to the bottom (i.e., the beginning of the ArrayList)  
 */  
public void addCard(Card c)  
{  
    deck.add(0, c);  
}  
}
```

shuffle works by swapping
each Card entry with another
randomly selected Card

deal a card from the top (the
end of the list)

add a card at the bottom (the
front of the list)

13

Dealing cards: silly examples

```
public class Dealer  
{  
    private DeckOfCards deck;  
  
    public Dealer()  
    {  
        deck = new DeckOfCards();  
        deck.shuffle();  
    }  
  
    public String dealTwo()  
    {  
        Card card1 = deck.dealCard();  
        Card card2 = deck.dealCard();  
  
        String message = card1 + " " + card2 ;  
  
        if (card1.getRank() == card2.getRank()) {  
            message += ": IT'S A PAIR";  
        }  
        return message;  
    }  
  
    public String dealHand(int numCards)  
    {  
        ArrayList<Card> hand = new ArrayList<Card>();  
        for (int i = 0; i < numCards; i++) {  
            hand.add(deck.dealCard());  
        }  
  
        return hand.toString();  
    }  
}
```

constructor creates a
randomly shuffled deck

dealTwo deals two cards
from a deck and returns a
message based on the cards

dealHand deals a specified
number of cards into an
ArrayList, returns their String
representation

14

Dealing cards: silly examples

```
...  
public void dealAndAsk()  
{  
    Scanner input = new Scanner(System.in);  
  
    String response = "yes";  
    while (!response.equals("no")) {  
        System.out.println(deck.dealCard());  
  
        System.out.print("Another? (yes/no):");  
        response = input.next();  
    }  
}
```

can deal cards based on user input, using a `Scanner` object to read from the console (`System.in`)
will keep dealing and displaying cards until the user says "no"

what if the user types "NO" ?
generalize?

what if the deck is exhausted?
generalize?

15

HW6

you are to define a `Skip3` class for playing the game

- start with shuffled deck and empty row (`ArrayList?`) of cards
- allow the player to:
 - deal a card, placing it at the end of the row
 - OR
 - move a card on top of a matching card 1 or 3 spots to the left
- if the user attempts an illegal deal (the deck has been exhausted or there are moves that can be made), then an error message should be displayed and the deal attempt ignored
- if the user attempts an illegal move (cards don't match, not 1 or 3 spots), then an error message should be displayed and the move attempt ignored
- the row of cards should be displayed after each deal/move
- the user should also be able to access the length of the row and the number of cards remaining in the deck

16

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
e.g., 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

so, why use arrays?

1st answer: DON'T! when possible, take advantage of ArrayList's flexibility

2nd answer: arrays can store primitives directly, without autoboxing/unboxing
as we saw earlier, there are some subtleties involved with autoboxing/unboxing

3rd answer: initializing/accessing an array can be easier in some circumstances

17

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;
}
```

18

Letter frequency

suppose we wanted to extend our DocumentFreq class to also keep track of letter frequencies

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
- could have an ArrayList of 26 counters, but would need to add each value and rely on autoboxing/unboxing
- instead, have an array of 26 counters
 - letters[0] is the counter for 'a'
 - letters[1] is the counter for 'b'
 - ...
 - letters[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%) |

19

Adding letter frequency to DocumentFreq

```
public class DocumentFreq
{
    . . .

    private int totalLetterCount;
    private int[] letters;

    /**
     * Reads in a text file and stores individual words and letter frequencies.
     * @param filename the text file to be processed (assumed to be in project folder)
     */
    public DocumentFreq(String filename) throws java.io.FileNotFoundException
    {
        totalLetterCount = 0;
        letters = new int[26];

        Scanner infile = new Scanner(new File(filename));
        while (infile.hasNext()) {
            String nextStr = infile.next();
            nextStr = nextStr.toLowerCase();
            nextStr = StringUtils.stripNonLetters(nextStr);

            totalLetterCount += nextStr.length();

            for (int i = 0; i < nextStr.length(); i++) {
                letters[nextStr.charAt(i) - 'a']++;
            }
        }
    }
}
```

by default, ints in an array are initialized to 0 (& doubles to 0.0)

you can subtract characters to get the difference in the ordering:

```
'a' - 'a' == 0
'b' - 'a' == 1
...
'z' - 'a' == 25
```

20

Adding letter frequency to DocumentFreq (cont.)

```

    . . .

    /**
     * @return total number of letters in the file
     */
    public int getTotalLetters()
    {
        return totalLetterCount;
    }

    /**
     * Displays the frequencies (and percentages) for all letters.
     */
    public void showLetters()
    {
        for (int i = 0; i < letters.length; i++) {
            System.out.printf("%c: %5d (%4.1f%%)\n", ('a'+i), letters[i],
                (100.0*letters[i]/totalLetterCount));
        }
    }
}

```

an array has a public field, `length`, which stores the capacity of the array

note: this is the number of spaces allotted for the array, not necessarily the number of items assigned

21

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%) |

22

Why arrays?

general rule: ArrayLists are better, more abstract, arrays – 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

23