CSC 221: Computer Programming I

Fall 2004

ArrayLists and OO design

- ArrayList summary
- example: word frequencies
- object-oriented design issues
  cohesion & coupling
- example: card game
- alternatives: arrays, wrapper classes

ArrayList methods

we have already seen:

`Object get(int index)`	returns object at specified index
`Object add(Object obj)`	adds obj to the end of the list
`Object add(int index, Object obj)`	adds obj at index (shifts to right)
`Object remove(int index)`	removes object at index (shifts to left)
`int size()`	removes number of entries in list
`boolean contains(Object obj)`	returns true if obj is in the list

other useful methods:

`Object set(int index, Object obj)`	sets entry at index to be obj
`int indexOf(Object obj)`	returns index of obj in the list (assumes obj has an `equals` method)
`String toString()`	returns a String representation of the list e.g., "[foo, bar, biz, baz]"

toString

the toString method is especially handy

- if a class has a toString method, it will automatically be called when printing or concatenating Strings

```
ArrayList words = new ArrayList();
words.add("one");
words.add("two");
words.add("three");

System.out.println(words);          // toString method is automatically
                                    // called to convert the ArrayList
                                    // → displays "[one, two, three]"
```

- when defining a new class, you should always define a toString method to make viewing it easier

Another example: word frequencies

recall the FileStats application

- read in words from a file, stored in a list, reported # of words and # of unique words

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 FileStats except must keep frequency counts

```
while (STRINGS REMAIN TO BE READ) {
    word = NEXT WORD IN FILE;
    word = word.toLowercase();

    totalWordCount++;

    if (ALREADY STORED IN LIST) {
        INCREMENT THE COUNT FOR THAT WORD;
    }
    else {
        ADD TO LIST WITH COUNT = 1;
}
```

Storage options

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

- `count[i]` is the number of times `word[i]` appears
- known as *parallel lists*

POTENTIAL PROBLEMS:

- have to keep the indices straight
- technically, can't store primitive types in an ArrayList
  (workarounds exist: arrays, wrapper classes – LATER)

BETTER YET:

- encapsulate the data and behavior of a word frequency 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

WordFreq class

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

    public WordFreq(String newWord)
    {
        word = newWord;
        count = 1;
    }

    public String getWord()
    {
        return word;
    }

    public int getFrequency()
    {
        return count;
    }

    public void increment()
    {
        count++;
    }

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

a `WordFreq` 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 `WordFreq` object

WordStats class

words stores `WordFreq` objects

constructor reads in words from the file – similar to `FileStats`

as each word is read in, the `ArrayList` is searched to see if it is already stored – if so, its count is incremented; if not, it is added (i.e., a `WordFreq` object for that word is added)

```
import java.io.*;
import java.util.ArrayList;

public class WordStats
{
    private int totalWordCount;
    private ArrayList words;

    public WordStats(String filename) throws FileNotFoundException
    {
        totalWordCount = 0;
        words = new ArrayList();

        FileScanner reader = new FileScanner(filename);
        while (reader.hasNext()) {
            String word = reader.nextString();
            word = word.toLowerCase();

            totalWordCount++;

            boolean found = false;
            for (int i = 0; i < words.size() && !found; i++) {
                WordFreq entry = (WordFreq)words.get(i);
                if (word.equals(entry.getWord())) {
                    entry.increment();
                    found = true;
                }
            }

            if (!found) {
                words.add(new WordFreq(word));
            }
        }
    }

    . . .
```


WordStats class (cont.)

`getTotalWords` and `getUniqueWords` are the same as in `FileStats`

`showWords` traverses the `ArrayList` and displays each word & frequency

note: implicitly makes use of the `toString` method

```
    . . .

    public int getTotalWords()
    {
        return totalWordCount;
    }

    public int getUniqueWords()
    {
        return words.size();
    }

    public void showWords()
    {
        for (int i = 0; i < words.size(); i++) {
            WordFreq entry = (WordFreq)words.get(i);
            System.out.println(entry);
        }
    }
}
```

Object-oriented design

our design principles so far:

- a **class** should model some entity, encapsulating all of its state and behaviors
  e.g., Circle, Die, Dot, AlleyWalker

- a **method** should implement one behavior of an object
  e.g., moveLeft, moveRight, getDiameter, draw, erase, changeColor

- a **field** should store some value that is part of the state of the object (and which must persist between method calls)
  e.g., xPosition, yPosition, color, diameter, isVisible
- 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

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

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

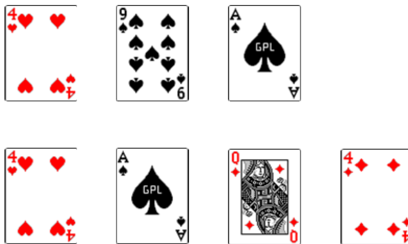
advantages of loose coupling:

- 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

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



Skip-3 design

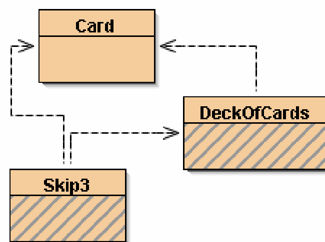
what are the entities involved in the game that must be modeled?

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

which entity relies upon another? how do they interact?

note: when you define classes in BlueJ, coupling is automatically diagrammed using dashed arrows

when you edit and compile a class, other classes that depend on it are marked to be recompiled



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 deck;

    public DeckOfCards() { … }

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

    public int cardsRemaining() { … }
}
```

Card class

```
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

```
    . . .

    /**
     * Converts the Card object into a String (e.g., for printing).
     *   @return a string consisting of rank followed by suit (e.g., "2H" or "QD")
     */
    public String toString()
    {
        return "" + rank + suit;
    }


    /**
     * Comparison method
     *   @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;
        }
    }
}
```

DeckOfCards class

```
public class DeckOfCards
{
    private ArrayList deck;

    /**
     * Creates a deck of 52 cards in order.
     */
    public DeckOfCards()
    {
        deck = new ArrayList();

        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();
    }

   . . .
```


DeckOfCards class

```
    . . .

    /**
     * Shuffles the deck so that the locations of the cards are random.
     */
    public void shuffle()
    {
        Random randy = new Random();

        for (int c = deck.size()-1; c > 0; c--) {
            int swapIndex = randy.nextInt(c);

            Card tempCard = (Card)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 (Card)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);
    }
}
```

utilizes the `Random` class from `java.util.Random`

the `nextInt` method returns a random integer in the range 0..(c-1)

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 hand = new ArrayList();
        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

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 to deal a card after the deck has been exhausted, 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

ArrayLists and primitives

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., each item in the array must be of the same type

  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 have the advantage that they can store primitive values

e.g., want to process a file and keep track of the frequency for each letter

can't store counts in an ArrayList*, but can store them in an array

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];
```

to access an item in an array, use brackets containing the desired index

- similar to `get` and `set` methods of ArrayList
- but, no need to cast when accessing an item from the array

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

Letter frequency

suppose we wanted to extend our WordStats 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
- 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  (9.217046580773042%)
b: 12  (1.1892963330029733%)
c: 28  (2.7750247770069376%)
d: 49  (4.856293359762141%)
e: 151  (14.965312190287413%)
f: 21  (2.0812685827552033%)
g: 23  (2.2794846382556986%)
h: 65  (6.442021803766105%)
i: 59  (5.847373637264618%)
j: 0  (0.0%)
k: 2  (0.19821605550049554%)
l: 39  (3.865213082259663%)
m: 14  (1.3875123885034688%)
n: 71  (7.036669970267591 4%)
o: 80  (7.928642220019822%)
p: 15  (1.4866204162537167%)
q: 1  (0.09910802775024777%)
r: 70  (6.937561942517344%)
s: 36  (3.5678889990089195%)
t: 109  (10.802775024777008%)
u: 15  (1.4866204162537167%)
v: 20  (1.9821605550049555%)
w: 26  (2.576808721506442%)
x: 0  (0.0%)
y: 10  (0.9910802775024777%)
z: 0  (0.0%)
```


Adding letter frequency to WordStats

```
public class WordStats
{
    . . .

    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 WordStats(String filename) throws FileNotFoundException
    {
        totalLetterCount = 0;
        letters = new int[26];

        FileScanner reader = new FileScanner(filename);
        while (reader.hasNext()) {
            String word = reader.nextString();
            word = word.toLowerCase();

            for (int i = 0; i < word.length(); i++) {
                if (Character.isLetter(word.charAt(i))) {
                    letters[word.charAt(i)-'a']++;
                    totalLetterCount++;
                }
            }

            . . .
        }
    }
```

note: if the character is not a letter, ignore it

you can subtract characters to get the difference in the ordering:
'a' – 'a' == 0
'b' – 'a' == 1

Adding letter frequency to WordStats (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.println((char)('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 the number of actual items assigned

Wrapper classes

UGLY alternative: there is a workaround that allows you to store primitive values in an ArrayList

- *wrapper classes* (`Integer`, `Double`, `Character`) exists to "wrap" primitive values up inside objects, which can then be stored
- methods (`intValue`, `doubleValue`, `charValue`) allow you to retrieve ("unwrap") the primitive value stored in the object

```
ArrayList numbers = new ArrayList;

Integer intObject = new Integer(5);         // wraps 5 inside an Integer

numbers.add(intObject);                     // can store the object

Integer obj = (Integer)numbers.get(0);      // can retrieve the object
int x = obj.intValue();                     // and access int value
```

Comparing a wrapper implementation

```
private int[] letters;

letters = new int[26];
for (int i = 0; i < letters.length; i++) {
    letters[i] = 0;
}
```

--- vs. ---

```
private ArrayList letters;

letters = new ArrayList();
for (int i = 0; i < 26; i++) {
    letters.add(new Integer(0));
}
```

```
letters[word.charAt(i)-'a']++;
```

--- vs. ---

```
lettersList.set(word.charAt(i)-'a',
                new Integer(((Integer)lettersList.get(word.charAt(i)-'a')).intValue() + 1));
```

```
System.out.println((char)('a'+i) + ": " +  letters[i] + "(" +
                    (100.0*letters[i]/totalLetterCount) + "%)");
```

--- vs. ---

```
System.out.println((char)('a'+i) + ": " + ((Integer)lettersList.get(i)).intValue() + "(" +
                   (100.0*((Integer)lettersList.get(i)).intValue()/totalLetterCount) + "%)");
```


Wrappers vs. arrays vs. OO

using wrapper classes to store primitives is UGLY

- having to create objects, cast when accessing, extract primitive value is tedious and error-prone
- Java 5.0 (released in 10/04) makes some of this automatic, but still tricky

arrays are a cleaner alternative if you can get along without the flexibility

better yet, avoid situations where you store collections of primitives

- a more object-oriented approach would be to define a class that encapsulates all the state and behaviors of letter counts

e.g., define a `LetterFreq` class that encapsulates a letter and its frequency
similar to `WordFreq`, this class would provide a method for incrementing
can then store and access `LetterFreq` objects in an ArrayList