CSC 221: Computer Programming I

Fall 2006

Lists, data storage & access

- ArrayList class
  - methods: add, get, size, remove, contains, set, indexOf, toString
- example: Notebook
- example: DeckOfCards
- HW6: Skip-3 solitaire
- user input, Scanner class

1

Composite data types

String is a composite data type

- each String object represents a collection of characters in sequence
- can access the individual components & also act upon the collection as a whole

many applications require a more general composite data type, e.g.,

- ✓ a to-do list will keep track of a sequence/collection of notes
- ✓ a dictionary will keep track of a sequence/collection of words
- ✓ a payroll system will keep track of a sequence/collection of employee records

Java provides several library classes for storing/accessing collections of arbitrary items

2

ArrayList class

an `ArrayList` is a generic collection of objects, accessible via an index

- must specify the type of object to be stored in the list
- create an `ArrayList<?>` by calling the `ArrayList<?>` constructor (no inputs)

```
ArrayList<String> words = new ArrayList<String>();
```

- add items to the end of the `ArrayList` using `add`

```
words.add("Billy");          // adds "Billy" to end of list
words.add("Bluejay");        // adds "Bluejay" to end of list
```

- can access items in the `ArrayList` using `get`
  - similar to Strings, indices start at 0

```
String first = words.get(0);    // assigns "Billy"
String second = words.get(1);   // assigns "Bluejay"
```

- can determine the number of items in the `ArrayList` using `size`

```
int count = words.size();     // assigns 2
```

3

Simple example

```
ArrayList<String> words = new ArrayList<String>();

words.add("Nebraska");
words.add("Iowa");
words.add("Kansas");
words.add("Missouri");

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

since an `ArrayList` is a composite object, we can envision its representation as a sequence of indexed memory cells

"Nebraska"	"Iowa"	"Kansas"	"Missouri"
0	1	2	3

exercise:

- given an `ArrayList` of state names, output index where "Hawaii" is stored

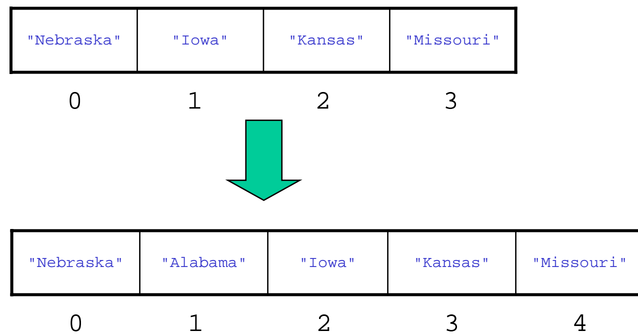
4

Other ArrayList methods: add at index

the generic `add` method adds a new item at the end of the `ArrayList`

a 2-parameter version exists for adding at a specific index

```
words.add(1, "Alabama");   // adds "Alabama" at index 1, shifting
                           // all existing items to make room
```



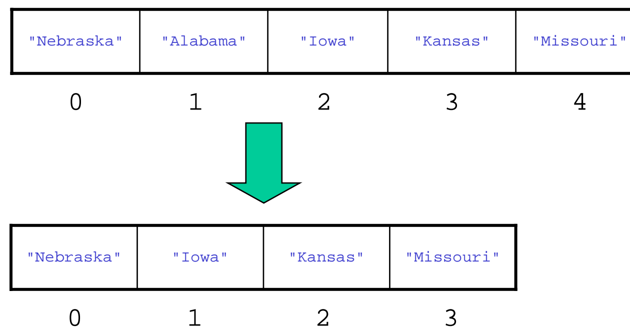
5

Other ArrayList methods: remove from index

in addition, you can remove an item using the `remove` method

- specify the item to be removed by index
- all items to the right of the removed item are shifted to the left

```
words.remove(1);
```



6

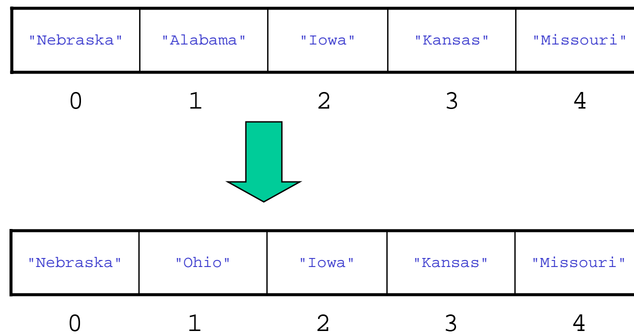
Other ArrayList methods: set at index

if you want to replace an item, you could remove then add

```
words.remove(1);
words.add(1, "Ohio");
```

or, more efficiently, call the `set` method

```
words.set(1, "Ohio");
```



7

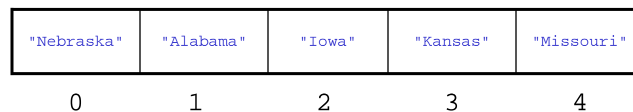
Other ArrayList methods: indexOf & toString

the `indexOf` method will search for and return the index of an item

- if the item occurs more than once, the first (smallest) index is returned
- if the item does not occur in the ArrayList, the method returns -1

```
words.indexOf("Kansas") → 3

words.indexOf("Alaska") → -1
```



0 1 2 3 4

the `toString` method returns a String representation of the list

- items enclosed in [ ], separated by commas

```
words.toString() → "[Nebraska, Alabama, Iowa, Kansas, Missouri]"
```

- the `toString` method is automatically called when printing an ArrayList

```
System.out.println(words) ≡ System.out.println(words.toString())
```

8

Notebook class

consider designing a class to model a notebook (i.e., a to-do list)

- will store notes in an `ArrayList<String>`
- will provide methods for adding notes, viewing the list, and removing notes

```
import java.util.ArrayList;

public class Notebook {
    private ArrayList<String> notes;

    public Notebook() { … }

    public void storeNote(String newNote) { … }
    public void storeNote(int priority, String newNote) { … }

    public int numberOfNotes() { … }

    public void listNotes() { … }

    public void removeNote(int noteNumber) { … }
    public void removeNote(String note) { … }
}
```

any class that uses an `ArrayList` must load the library file that defines it

9

Notebook class (cont.)

```
public class Notebook {
    private ArrayList<String> notes;

    /**
     * Constructs an empty notebook.
     */
    public Notebook() {
        this.notes = new ArrayList<String>();
    }

    /**
     * Store a new note into the notebook.
     *   @param  newNote  note to be added to the notebook list
     */
    public void storeNote(String newNote) {
        this.notes.add(newNote);
    }

    /**
     * Store a new note into the notebook with the specified priority.
     *   @param  priority  1 <= priority <= numberOfNotes()
     *   @param  newNote  note to be added to the notebook list
     */
    public void storeNote(int priority, String newNote) {
        this.notes.add(priority-1, newNote);
    }

    /**
     *   @return the number of notes currently in the notebook
     */
    public int numberOfNotes() {
        return this.notes.size();
    }

    . . .
```

constructor creates the (empty) `ArrayList`

one version of `storeNote` adds a new note at the end

another version adds the note at a specified index

`numberOfNodes` calls the `size` method

10

Notebook class (cont.)

```
    . . .

    /**
     * Show a note.
     *   @param notePriority the number of the note to be shown (first note is #1)
     *   @return true if note was shown (i.e., the index was valid)
     */
    public boolean showNote(int notePriority) {
        if (notePriority <= 0 || notePriority > this.numberOfNotes()) {
            return false;
        }
        else {
            String entry = this.notes.get(notePriority-1);
            System.out.println(entry);
            return true;
        }
    }

    /**
     * List all notes in the notebook.
     */
    public void listNotes() {
        System.out.println("NOTEBOOK CONTENTS");
        System.out.println("-----------------");
        for (int i = 1; i <= this.numberOfNotes(); i++) {
            System.out.print(i + ": ");
            this.showNote(i);
        }
    }

    . . .
```

`showNote` checks to make sure the note number is valid, then calls the `get` method to access the entry

`listNotes` traverses the `ArrayList` and shows each note (along with its #)

Notebook class (cont.)

```
    . . .

    /**
     * Removes a note.
     *   @param notePriority the number of the note to be removed (first is #1)
     *   @return true if the note was removed (i.e., the index was valid)
     */
    public boolean removeNote(int notePriority) {
        if (notePriority <= 0 || notePriority > this.numberOfNotes()) {
            return false;
        }
        else {
            this.notes.remove(notePriority-1);
            return true;
        }
    }

    /**
     * Removes a note.
     *   @param note the note to be removed
     *   @return true if the note was removed (i.e., it was in the list)
     */
    public void removeNote(String note) {
        int index = this.notes.indexOf(note);
        if (index == -1) {
            return false;
        }
        else {
            this.notes.remove(index);
            return true;
        }
    }
}
```

one version of `removeNote` takes a note #, calls the `remove` method to remove the note with that number

another version takes the text of the note and calls the `indexOf` method to search for it (and remove if found)

In-class exercises

download Notebook.java and try it out

- add notes at end
- add notes at beginning and/or middle
- remove notes by index
- remove notes by text

add a method that allows for assigning a random job from the notebook

- i.e., pick a random note, remove it from notebook, and return the note

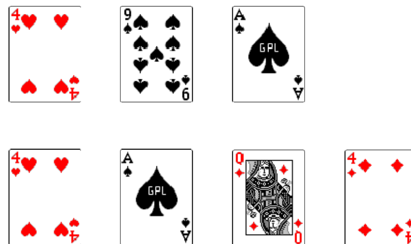
```
/**
 * @return a random note from the Notebook (which is subsequently removed)
 */
public String handleRandom() {


}
```


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

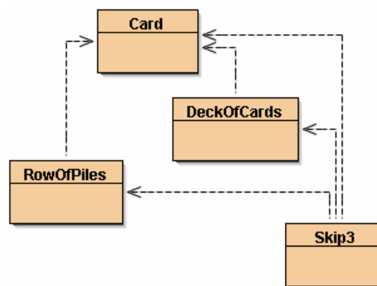


Skip-3 design

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?



Card & DeckOfCards

```
public class Card {
    private String cardStr;

    public Card(String cardStr) { … }

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

    public boolean matches(Card other) { … }
    public boolean equals(Object other) { … }

    public String toString() { … }
}
```

`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() { … }
}
```

Card class

```
public class Card {
    private String cardStr;

    /**
     * Creates a card with the specified rank and suit
     *   @param cardStr a two character String, where the first char is the rank
     *                  ('2', '3', '4', ..., '9', 'T', 'J', 'Q', 'K', or 'A') and
     *                  the second char is the suit ('S', 'H', 'D', or 'C')
     */
    public Card(String cardStr) {
        this.cardStr = cardStr.toUpperCase();
    }

    /**
     * @return the rank of the card (e.g., '5' or 'J')
     */
    public char getRank() {
        return this.cardStr.charAt(0);
    }

   /**
     * @return the suit of the card (e.g., 'S' or 'C')
     */
    public char getSuit() {
        return this.cardStr.charAt(1);
    }

    /**
     * @return a string consisting of rank followed by suit (e.g., "2H" or "QD")
     */
    public String toString() {
        return this.cardStr;
    }

    . . .
```

when constructing a Card, specify rank & suit in a String, e.g.,

`Card c = new Card("JH");`

accessor methods allow you to extract the rank and suit

17

Card class (cont.)

```
    . . .

    /**
     * @return true if other is a Card with the same rank OR suit, else false
     */
    public boolean matches(Card other) {
        return (this.getRank() == other.getRank() ||
                this.getSuit() == other.getSuit());
    }

    /**
     * @return true if other is a Card with the same rank AND suit, else false
     */
    public boolean equals(Object other) {
        return (this.getRank() == ((Card)other).getRank() &&
                this.getSuit() == ((Card)other).getSuit());
    }
}
```

for esoteric reasons, the parameter to the `equals` method must be of type `Object` (the generic type that encompasses all object types)

- must then cast the `Object` into a `Card`

18

DeckOfCards class

```
import java.util.ArrayList;
import java.util.Collections;

public class DeckOfCards {
    private ArrayList<Card> deck;

    /**
     * Creates a deck of 52 cards in order.
     */
    public DeckOfCards() {
        this.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));
                this.deck.add(c);
            }
        }
    }

    /** Adds a card to the bottom of the deck.
     *   @param c the card to be added to the bottom (i.e., beginning of the ArrayList)
     */
    public void addCard(Card c) {
        this.deck.add(0, c);
    }

    /** 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 this.deck.remove(this.deck.size()-1);
    }

   . . .
```

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

add a card at the front (index 0)

deal from the end (index size()-1)

19

DeckOfCards class

```
    . . .

    /**
     * Shuffles the deck so that the locations of the cards are random.
     */
    public void shuffle() {
    {
        Collections.shuffle(this.deck);
    }

    /** @return the number of cards remaining in the deck
     */
    public int cardsRemaining() {
        return this.deck.size();
    }

    /** @return a String representation of the deck
     */
    public String toString() {
        return this.deck.toString();
    }
}
```

the `Collections` class contains a static method for randomly shuffling a list

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

the `toString` method simply returns the string representation of the underlying ArrayList, e.g.,

`"[3D, TH,  AH, 9S, JC, 4C]"`

20

Dealing cards: silly examples

```
import java.util.ArrayList;

public class Dealer {
    private DeckOfCards deck;

    public Dealer() {
        this.deck = new DeckOfCards();
        this.deck.shuffle();
    }

    public void dealTwo() {
        Card card1 = this.deck.dealCard();
        Card card2 = this.deck.dealCard();

        System.out.println(card1 + " " + card2);

        if (card1.getRank() == card2.getRank()) {
            System.out.println("IT'S A PAIR");
        }
    }

    public ArrayList<Card> dealHand(int numCards) {
        ArrayList<Card> hand = new ArrayList<Card>();
        for (int i = 0; i < numCards; i++) {
            hand.add(this.deck.dealCard());
        }

        return hand;
    }
}
```

constructor creates a randomly shuffled deck

`dealTwo` deals two cards from a deck and displays them (also identifies a pair)

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

21

HW6: Skip-3 solitaire

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

- `RowOfPiles()`: constructs an empty row (ArrayList of Cards)
- `void addPile(Card top)`: adds a new pile to the end of the row
- `boolean movePile(Card fromTop, Card toTop)`: moves one pile on top of another, as long as 1 or 3 spots away
- `int numPiles()`: returns number of piles in the row
- `String toString()`: returns String representation of the row

a simple `Skip3` class, which utilizes a `RowOfPiles` to construct an interactive game, is provided for you

- you will make some improvements (error messages, card counts, etc.)

22

```java
import java.util.Scanner;

public class Skip3 {
    private DeckOfCards deck;
    private RowOfPiles row;

    public Skip3() {
        this.restart();
    }

    public void restart() {
        this.deck = new DeckOfCards();
        this.deck.shuffle();
        this.row = new RowOfPiles();
    }

    public void playGame() {
        Scanner input = new Scanner(System.in);

        boolean gameOver = false;
        while (!gameOver) {
            System.out.println(this.row);
            System.out.print("Action? ");

            char response = input.next().toLowerCase().charAt(0);
            if (response == 'd') {
                if (this.deck.cardsRemaining() > 0) {
                    this.row.addPile(this.deck.dealCard());
                }
            }
            else if (response == 'm') {
                String from = input.next().toUpperCase();
                String to = input.next().toUpperCase();
                this.row.movePile(new Card(from), new Card(to));
            }
            else if (response == 'e') {
                gameOver = true;
            }
        }
    }
}
```

Skip3

`Skip3` class utilizes a `DeckOfCards` and a `RowOfPiles`

a `Scanner` object is used to read commands from the user

`new Scanner(System.in)`

specifies input is to come from the keyboard

the `next()` method reads the next String (delineated by whitespace) entered by the user

23