Mat 2170
Week 14

ArrayList Class

Spring 2014

Student Responsibilities

- **EXAM** – Thursday, 4/24, 7:00 pm
  one sheet of 8.5" by 11" paper for notes is allowed
- Reading: Textbook, Chapter 11
- Lab: ArrayList and more on writing classes from scratch
- Attendance

Lab 14: One Handed Solitaire – An OverView

- A standard **Deck** of 52 cards, shuffled (with user's seed)
- Play continues until **Deck** is empty
- When **Hand** is empty, deal from "top," otherwise deal from "bottom"
- After a **Card** is dealt, "collapse" **Hand** (if possible), comparing the "top" **Card**, and the one 3 **Card**s back from it
  - If the **Rank**s match, discard the 4 top **Card**s in the **Hand**
  - If the **Suit**s match, discard the two **Card**s after the top **Card**
- Continue collapsing until no more matches
- Score is number of **Card**s left in **Hand** at the end of the game
- Program repeats until user enters -1 for the shuffle seed

Class Card

- Data members:
  - suit
  - rank
  - (faceup isn't needed for this game)
- Member methods:
  - constructor()
  - suitsMatch(), ranksMatch()
  - toString(), quickString()

Class Deck

- Data member:
  - an ArrayList of Card
- Member methods:
  - Two constructors
  - shuffle(seed)
  - isEmpty(), size()
  - dealTop(), dealBottom()
  - getCard(), add(), remove()
  - toString()

The `Deck` Shuffle

```
public void shuffle(int seed)
 {
     Collections.shuffle(deck, new Random(seed));
 }
```

Where `deck` is the name of the `ArrayList` in `Deck` class.

Use: `import java.util.*`

The Collapse

```
If there are at least 4 cards in the Hand:

    Create and initialize currentCard (on top)
         and matchCard (3 back)

    While there are at least 4 cards in the Hand,
          and either ranks or suits match:

       If ranks match,
            delete the top 4 cards
       Else if suits match,
            delete the 2 below the top card

       If Hand has at least 4 cards
            re-initialize currentCard and matchCard
```


Lab 14

Questions?

The ArrayList Class

- The **java.util** package includes a **class** called **ArrayList** that extends the usefulness of arrays by providing additional operations.
- Since **ArrayList** is a class, all operations on **ArrayList** objects are indicated using method calls.
- In the summary of **ArrayList** methods which follows, the notation <**T**> indicates the **base type** of the `ArrayList` object.

ArrayList Methods

`boolean add(<T> element)`
Adds a new element to the end of the `ArrayList`; the return value is always `true`

`void add(int index, <T> element)`
Inserts a new element into the `ArrayList`; **before** the position specified by `index`

`<T> remove(int index)`
Removes the element at the specified position and returns that value

`boolean remove(<T> element)`
Removes the first instance of `element`, if it appears; returns `true` if a match is found

`void clear()`
Removes all elements from the `ArrayList`

`int size()`
Returns the number of elements in the `ArrayList`

`<T> get(int index)`
Returns the object at the specified index

`<T> set(int index, <T> value)`
Sets the element at the specified index to the new value and returns the old value

`indexOf(<T> value)`
Returns the index of the first occurrence of the specified value, or -1 if it does not appear

`boolean contains(<T> value)`
Returns `true` if the `ArrayList` contains the specified value

`boolean isEmpty()`
Returns `true` if the `ArrayList` contains no elements

Generic Types in Java

- The **type parameter** $< T >$ used in the previous slides is a placeholder for the **element type** used in the array.
- Class definitions that include a **type parameter** are called **generic** types.
- When we declare or create an `ArrayList`, it is a good idea to specify the element type in **angle brackets**. For example:

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

- This allows Java to check for the correct element type when `set()` is called, and eliminates the need for a type cast when `get()` is called.

- Java includes a wrapper class to correspond to each of the primitive types:

`boolean`	↔	`Boolean`	`float`	↔	`Float`
`byte`	↔	`Byte`	`int`	↔	`Integer`
`char`	↔	`Character`	`long`	↔	`Long`
`double`	↔	`Double`	`short`	↔	`Short`

- The value stored in the object `maxItems` is an object, and we can use it in any context that require objects.

Using Wrapper Classes

- All of the primitive wrapper classes in Java are **immutable** – their states cannot be modified after they are created.
- For each wrapper class, Java defines a method to retrieve the primitive value, e.g.:

```
int underlyingValue = maxItems.intValue();
```

- Java will automatically **box** and **unbox** the primitive values in a wrapper class.

Generic Types and Boxing/Unboxing

- Automatic conversion of values between a primitive type and the corresponding wrapper class allows an `ArrayList` object to store primitive values, even though the elements of any `ArrayList` must be a Java class.
- For example:

```
ArrayList <Integer> myList = new ArrayList<Integer>();
myList.add(42);
int answer = myList.get(0);
```

In the second statement, Java uses **boxing** to enclose 42 in a wrapper object of type `Integer`; the third statement **unboxes** the `Integer` to obtain the `int`.

Reversing an `ArrayList`

```
import acm.program.*;
import java.util.*;

public class ReverseArrayList extends ConsoleProgram {
   public void run()
   {
      println("This program reverses the elements " +
              "in an ArrayList.");
      println("Use " + SENTINEL + " to signal the " +
              "end of the list.");

      ArrayList<Integer> myList = readIntArrayList();
      reverseArrayList(myList);
      printIntArrayList(myList);
   }
```


`readIntArrayList()`

```
/* Reads the data into the list */
private ArrayList<Integer> readIntArrayList()
{
   ArrayList<Integer> list = new ArrayList<Integer>();

   int value = readInt(" ? ");
   while (value != SENTINEL)
   {
      list.add(value);
      value = readInt(" ? ");
   }

   return list;
}


/* Private constant --- Define the end-of-data value */
private static final int SENTINEL = 0;
```

reverseArrayList() & swapElements()

```
/* Reverses the data in an ArrayList */
private void reverseArrayList(ArrayList<Integer> list)
{
   for (int i = 0; i < list.size() / 2; i++)
   {
      swapElements(list, i, list.size() - i - 1);
   }
}


/* Exchanges two elements in an ArrayList */
private void swapElements(ArrayList<Integer> list,
                          int p1, int p2)
{
   int temp = list.get(p1);
   list.set(p1, list.get(p2));
   list.set(p2, temp);
}
```


ArrayList Searching Methods

Method	Description
`contains(value)`	returns `true` if the given value appears in the list Ex: `list.contains("hello")`
`indexOf(value)`	returns the index of the first occurrence of the given value in the list (-1 if not found) Ex: `list.indexOf("world")`
`lastIndexOf(value)`	returns the index of the last occurrence of the given value in the list (-1 if not found) Ex: `list.lastIndexOf("hello")`

Where **list** is an `ArrayList<string>`.

ArrayList Sorting and Binary Search Methods

The **java.util** package contains a class called **Collections** which contains several useful static methods. Of particular interest:

Method	Description
`sort(list)`	rearranges the elements into sorted (non–decreasing) order Ex: `Collections.sort(L)`
`binarySearch(list, value)`	searches a **sorted** list for a given element value and returns its index Ex: `Collections.binarySearch (L,"hello")`

When an **ArrayList** is sorted, **binary search** is **much** faster than **linear search**.