CS 367 Announcements
Thursday, January 22, 2015

Sign Up for Piazza

Last Time

Course Info
- website - http://pages.cs.wisc.edu/~cs367-1/
- ➢ see syllabus page for online readings and lecture outlines

Collections
- Bag Intro
- Abstract Data Types
- designing the Bag ADT - Java interfaces
- using the Bag ADT

Characteristics of Good & Reusable Software

Implementing the Bag ADT using Java **`Object`**s

Today

Course Topics (from last time)

Implementing the Bag ADT
- casting when using Object
- using Java generics for generality

List ADT
- coding the ListADT as a Java interface
- using lists via the ListADT

Next Time

Read: continue *Lists*

Lists
- implementing the ListADT using an array (SimpleArrayList)

Java API Lists

Iterators
- concept
- iterators and the Java API
- using iterators

Recall the Bag ADT

Bag ADT

is that really unordered?

A Bag is a general unordered container of items where duplicates are allowed.

```
import java.util.*;

public interface BagADT {

    void add(Object item);
    Object remove() throws NoSuchElementException;
    boolean isEmpty();
}
```

→ Why were we using the `Object` class in our `BagADT` interface?

Steps

1. design the ADT
   - concept
   - operations / issues
   - code this up with an interface

2. use the ADT
   - coding the application

3. implement it
   - code the class

Using `BagADT` and Casting

Instantiating a General Container Object

the form of instantiating a general container:

ADT_NAME	OBJECT_NAME	=	NEW	IMPLMETING_CLASS_NAME();
java interface	java identifier			java class for the specific impl we want to use

→ **Write a statement** that makes a *general* bag container named `bag`.

BagADT bag = new ArrayBag();

→ Assume `Die` is a class representing dice and has a zero parameter constructor.
Write a code fragment that adds 6 dice to `bag`.

```
for (int i = 0; i < 6; i ++)
    bag.add(new Die());
```

—

note: add expects type Object but gets type Die!
which is ok, but there is UPCASTING: specific type -> general type
e.g. die -> object (automatic)

→ Assume the bag has had items added to it. **Why doesn't the following code compile?**

```
while (!bag.isEmpty()) {
    Die myDie = bag.remove();
    myDie.roll();
}
```

→

```
Object obj = bag.remove();
if (obj.instanceOf Die) {
    Die myDie = (Die) obj;
    myDie.roll();
}
```

bag.remove() returns type Object
but myDie requires type Die

so it is not ok! It is called DOWNCASTING : general -> specific
Object -> Die

Different from upcasting, downcasting is not automatic

Java Generics - a Better Way to Make a General Bag ADT

What changes are needed to make the interface below generic?

```
import java.util.*;

public interface BagADT {

    void add(Object item);

    Object remove() throws NoSuchElementException;

    boolean isEmpty();

}
```

type parameter

<E>

E

E

JAVA GENERICS specify >= 1 type parameter!
- listed after interface / class name
- are between < >

The type parameters is used to replace the item type (e.g. Object)

How is a generic interface used to make a Bag?

- the declaration specifies the item type
- container can then only store that type of item
- No downcasting is required!

e.g.

BagADT <Die> bag = ……

- so this bag can only store die!

Using the Generic Bag ADT

How do we use a generic interface and its generic implementation?

→ **Write a code fragment** to make one generic Bag ADT
storing `String` objects and another one storing `Die` objects.

```
BagADT <String> bag1 = new ArrayBag <String> ( );
BagADT <Die> bag2 = new ArrayBag <Die> ( );
```

→ **Write a statement** to add "cs367" into the appropriate Bag ADT.

```
bag1.add("cs367");
```

→ Can we make a single generic Bag ADT
that can store both `String` and `Die` objects at the same time?

```
BagADT <Object> mixedBag = new ArrayBag( );
```

Still can make containers that store mixed types
but now down casting is required

Implementing the Generic `BagADT`

What changes are needed to make the implementation below generic?

```
public class ArrayBag < E > implements BagADT < E > {

   //instance variables
   private E [ ] items

   private int numItems;

   private static final …


   //constructor
   public ArrayBag() {

       // items = new Object [ INITIAL_CAPACITY ]; old version
       item = ( E [ ] ) new Object [ INITIAL_CAPACITY ] ; // this is right

       numItems = 0;



   }

   //BagADT methods

   public boolean isEmpty ( ) { … }


   public void add ( E item ) { … }


   public E remove ( ) throws … { … }



}
```

Design - List ADT

Concept

1. A general container
2. A contiguous (no gap, no space between items) collection
3. Position oriented with 0 - based indexing
4. Duplicates are allowed
5. Expand
6. shifting maintains the relative ordering

Operations

- add item at end of list

- add item at specified position — shifts other things to make room ! — 0 <= position <= size (!!!)
- get item at specified position — 0 <= position < size
- remove item at specified position — shifts to fill the gap — 0 <= position < size

- check if list contains a specified item

- get size of list (number of items it contains)
- check if list is empty

Issues

Null item – detect then signal with `IllegalArgumentException`

Bad position – detect then signal with `IndexOutOfBoundsException`

Empty list – handle as a bad position

Interface - Generic `ListADT`

```
/**
 * A List is a general container storing a contiguous collection
 * of items, that is position-oriented using zero-based indexing
 * and where duplicates are allowed.
*/
public interface ListADT <E> {

   /**
    * Add item to the end of the List.
    *
    * @param item the item to add
    * @throws IllegalArgumentException if item is null
    */
   void add(E item);

   /**
    * Add item at position pos in the List, moving the items
    * originally in positions pos through size()- 1 one place
    * to the right to make room.
    *
    * @param pos the position at which to add the item
    * @param item the item to add
    * @throws IllegalArgumentException if item is null
    * @throws IndexOutOfBoundsException if pos is less than 0
    * or greater than size()
    */
   void add(int pos, E item);

   /**
    * Return true iff item is in the List (i.e., there is an
    * item x in the List such that x.equals(item))
    *
    * @param item the item to check
    * @return true if item is in the List, false otherwise
    */
   boolean contains(E item);
```

Interface - Generic `ListADT` (cont.)

```
    /**
     * Return the number of items in the List.
     *
     * @return the number of items in the List
     */
    int size();

    /**
     * Return true iff the List is empty.
     *
     * @return true if the List is empty, false otherwise
     */
    boolean isEmpty();

    /**
     * Return the item at position pos in the List.
     *
     * @param pos the position of the item to return
     * @return the item at position pos
     * @throws IndexOutOfBoundsException if pos is less than 0
     * or greater than or equal to size()
     */
    E get(int pos);

    /**
     * Remove and return the item at position pos in the List,
     * moving the items originally in positions pos+1 through
     * size() one place to the left to fill in the gap.
     *
     * @param pos the position at which to remove the item
     * @return the item at position pos
     * @throws IndexOutOfBoundsException if pos is less than 0
     * or greater than or equal to size()
     */
    E remove(int pos);
}
```

Use - `ListADT`

→ Assume **`myList`** is a ListADT. **What does the following code fragment do in general?**

```
for (int i = 0; i < myList.size(); i++) {
   myList.remove(i);
}
```

It does NOT empty the list!

What is really does:

remove all the items originally in the even positions!

Use - `ListADT`

→ Assume `myList` is a ListADT. **Write a code fragment** to reverse `myList` without using any additional ListADTs or other data structures (e.g., array).

```
for (int i = 1; i < myList.size( ); i ++ ){
        myList.add( 0, myList.remove( i ) );
}
```

Does this work? -> YES!

Implement it!