List and ArrayList

Thursday, February 16, 2023 6:40 PM

ARRAY LISTS

An `ArrayList` provides an alternative way of storing a list of objects and has the following advantages over an array:

- An `ArrayList` shrinks and grows as needed in a program, whereas an array has a fixed length that is set when the array is created.
- In an `ArrayList` `list`, the last slot is always `list.size()-1`, whereas in a partially filled array, you, the programmer, must keep track of the last slot currently in use.
- For an `ArrayList`, you can do insertion or deletion with just a single statement. Any shifting of elements is handled automatically. In an array, however, insertion or deletion requires you to write the code that shifts the elements.

Notice1: the size of array list can shrink and grow

Collections and Generics

The collections classes are generic, with type parameters. Thus, `List<E>` and `ArrayList<E>` contain elements of type `E`.

When a generic class is declared, the type parameter is replaced by an actual object type. For example,

```
private ArrayList<Clown> clowns;
```

Notice2: Array list can only contain object but not primitive type

NOTE

1. The `clowns` list must contain only `Clown` objects. An attempt to add an `Acrobat` to the list, for example, will cause a compile-time error.
2. Since the type of objects in a generic class is restricted, the elements can be accessed without casting.
3. All of the type information in a program with generic classes is examined at compile time. After compilation the type information is erased. This feature of generic classes is known as *erasure*. During execution of the program, any attempt at incorrect casting will lead to a `ClassCastException`.

THE `List<E>` INTERFACE

A class that implements the `List<E>` interface—`ArrayList<E>`, for example—is a list of elements of type `E`. In a list, duplicate elements are allowed. The elements of the list are indexed, with `0` being the index of the first element.

A list allows you to

- Access an element at any position in the list using its integer index.
- Insert an element anywhere in the list.

The Methods of `List<E>`

Here are the methods you should know.

`boolean add(E obj)`

Appends `obj` to the end of the list. Always returns `true`. If the specified element is not of type `E`, throws a `ClassCastException`.

`int size()`

Returns the number of elements in the list.

`E get(int index)`

Returns the element at the specified `index` in the list.

`E set(int index, E element)`

Replaces item at specified `index` in the list with specified `element`. Returns the element that was previously at `index`. Throws a `ClassCastException` if the specified element is not of type `E`.

`void add(int index, E element)`

Inserts `element` at specified `index`. Elements from position `index` and higher have 1 added to their indices. Size of list is incremented by 1.

`E remove(int index)`

Removes and returns the element at the specified `index`. Elements to the right of position `index` have 1 subtracted from their indices. Size of list is decreased by 1.

Using `ArrayList<E>`

Example 1

```
//Create an ArrayList containing 0 1 4 9.
List<Integer> list = new ArrayList<Integer>(); //An ArrayList is-a List
for (int i = 0; i < 4; i++)
    list.add(i * i);  //example of auto-boxing
                      //i*i wrapped in an Integer before insertion
Integer intOb = list.get(2); //assigns Integer with value 4 to intOb.
                             //Leaves list unchanged.
int n = list.get(3);  //example of auto-unboxing
                      //Integer is retrieved and converted to int
                      //n contains 9
Integer x = list.set(3, 5);  //list is 0 1 4 5
                             //x contains Integer with value 9
x = list.remove(2);          //list is 0 1 5
                             //x contains Integer with value 4
list.add(1, 7);              //list is 0 7 1 5
list.add(2, 8);              //list is 0 7 8 1 5
```

Example 2

```
//Traversing an ArrayList of Integer.
//Print the elements of list, one per line.
for (Integer num : list)
    System.out.println(num);
```

Example 3

```
/** Precondition:  List list is an ArrayList that contains Integer
 *                 values sorted in increasing order.
 *  Postcondition: value inserted in its correct position in list.
 */
public static void insert(List<Integer> list, Integer value)
{
    int index = 0;
    //find insertion point
    while (index < list.size() &&
           value.compareTo(list.get(index)) > 0)
        index++;
    //insert value
    list.add(index, value);
}
```

Example 5

```
/** Swap two values in list, indexed at i and j. */
public static void swap(List<E> list, int i, int j)
{
    E temp = list.get(i);
    list.set(i, list.get(j));
    list.set(j, temp);
}
```

Example 6

```
/** Print all negatives in list a.
 *  Precondition:  a contains Integer values.
 */
public static void printNegs(List<Integer> a)
{
    System.out.println("The negative values in the list are: ");
    for (Integer i : a)
        if (i.intValue() < 0)
            System.out.println(i);
}
```

Example 7

```
/** Change every even-indexed element of strList to the empty string.
 *  Precondition:  strList contains String values.
 */
public static void changeEvenToEmpty(List<String> strList)
{
    boolean even = true;
    int index = 0;
    while (index < strList.size())
    {
        if (even)
            strList.set(index, "");
        index++;
        even = !even;
    }
}
```

Recall:

The `Integer` Class

The `Integer` class wraps a value of type `int` in an object. An object of type `Integer` contains just one instance variable whose type is `int`.

Here are the `Integer` methods you should know for the AP exam:

`Integer(int value)`

Constructs an `Integer` object from an `int`. (Boxing.)

`int compareTo(Integer other)`

Returns 0 if the value of this `Integer` is equal to the value of `other`, a negative integer if it is less than the value of `other`, and a positive integer if it is greater than the value of `other`.

`int intValue()`

Returns the value of this `Integer` as an `int`. (Unboxing.)

`boolean equals(Object obj)`

Returns `true` if and only if this `Integer` has the same `int` value as `obj`.

NOTE

1. This method overrides `equals` in class `Object`.
2. This method throws a `ClassCastException` if `obj` is not an `Integer`.

`String toString()`

Returns a `String` representing the value of this `Integer`.

Here are some examples to illustrate the `Integer` methods:

```
Integer intObj = new Integer(6);  //boxes 6 in Integer object
int j = intObj.intValue();        //unboxes 6 from Integer object

System.out.println("Integer value is " + intObj);
//calls toString() for intObj
//output is
//Integer value is 6
```

a