Generic, Custom Array Lists

Lists, array implementations, operations, encapsulation, computational complexity

Common list operations

A list is an ordered collection of elements.

Common list operations include:

- add – adds a new element to the list
- remove – removes an element from the list (and returns it)
- get – returns a list element (from a particular position or satisfying a particular condition)
- contains – determines if a particular item belongs to the list
- size – number of elements in the list

For many applications, ArrayList<E> or LinkedList<E>, which have generic type elements, provide all the needed list functionality.

What’s the problem?

Sometimes you need a custom list:

- To put restrictions on legal list elements (such as no null elements, no duplicate elements, etc).
- To support particular operations not common to all lists.
- To program efficiently for specific operations (such as add to front).
- To create “self organizing lists,” where the order changes dynamically according which elements are searched.
- … and many more.

Where to start?

Here’s the task. Create an array-based list with these features:

1. The underlying data structure is an array.
2. The list supports the common list operations:
    1. void add(E x, int index) // Adds x to the list at the given index.
    2. E remove(int index). // Removes x at the given list index
    3. int find(E x). // Find the index of x in the list
    4. int size(). // Number of list elements

Define a list interface for the operations

Create a list interface to declare the operations.

```
public interface IList<E> {
    void add(E x, int index);
    E remove(int index);
    E get(int index);
    int find(E x);
    int size();
}
```

Define list class attributes

Need 3 attributes:

- One for the capacity (number of array elements).
- One for the size (actual number of list elements).
- An array to hold the elements. Type should be E[].

```
public class SimpleArrayList<E> implements IList<E> {
   private int capacity = 100;
   private int size = 0;
   private E[] list;
```

Define the constructor

There will be two constructors:

- a null constructor
- one with an int parameter for the initial capacity

The null constructor simply calls the one with the int parameter. Note that constructors do not have <E> beside the class name.

```
public class SimpleArrayList<E> implements IList<E> {
    private int capacity = 100;
    private int size = 0;
    private E[] list;

    public SimpleArrayList( ) {
        this(capacity);
    }
```

Problems: array initialization

Here's the array declaration:

```
  private E[] list;
```

You cannot do this:

```
      list = new E[capacity];
      ERROR: Cannot create a generic array of E
```

Java needs to know the element type at compile time. But that won't be known until runtime.

We could try using Object[] instead, but this doesn't work either.

```
      list = new Object[capacity];
      ERROR: Cannot convert from Object[] to E[]
```

Fixing array initialization

Problems:

- *`Cannot create a generic array of E`*
- *`Cannot convert from Object[] to E[]`*

One solution: Create an Object[ ] and cast to type E[ ] in the constructor. (Could also do that in the declaration.)

```
public SimpleArrayList(int capacity) {
    if (capacity <= 0)
        throw new IllegalArgumentException();
    this.capacity = capacity;
    Object[] temp = new Object[capacity];
    list = (E[]) temp;
}
```

Implementing SimpleArrayList

Code for SimpleArrayList is analogous to code for StudentList. Here is the remove method.

```
public E remove(int index) {
   if (index < 0 || index >= size)
      throw new IllegalArgumentException();
   E value = list[index];
   for (int k = index; k < size - 1; k++)
      list[k] = list[k + 1];
   size--;
   list[size] = null;
   return value;
}
```

The assignment to null releases the old reference to the last item so it can be removed by Java's garbage collector.

Restricting element types

SimpleArrayList has a single type parameter <E>, so it can be used for *any* element type. But you may want a list to support an operation that is restricted to certain types.

The next example requires that the elements be some numeric type in order to perform arithmetic on them.

```
public class ListOfNumbers<E extends Number>
```

All elements are from a class that extends Number.

ListOfNumbers - constructor

The constructor should use the restricted type (Number) for the temporary array type.

```
public class ListOfNumbers<E extends Number> {
   private final int CAPACITY = 100;
   private int size = 0;
   private E[] list;

   public ListOfNumbers() {
      Number[] temp = new Number[CAPACITY];
      list = (E[]) temp;
   }
```

ListOfNumbers - operations

ListOfNumbers.average() requires list elements to support arithmetic operations.

```
public double average() {
   if (size == 0)
     return 0;
   double result = 0;
   for (int k = 0; k < size; k++) {
      result = result + list[k].doubleValue();
   }
   return result/size;
}
```