Generic ArrayLists

Reading: RS Chapter 15

Collections

- Data structures – stores elements in a manner that makes it easy for a client to work with the elements

- Specific collections are specialized for particular types of operations
  - Ordering
  - Duplicates
  - Add/remove

- Common set of operations: add, remove, get, contains, size
Lists

- List: an ordered collection of elements
  - Accessible by a zero-based index
  - Size: number of elements in the list
  - Elements may be added to an empty list, at the front, middle, and back
Scenario

• Suppose a Course has a list of enrolled students
• We want to enroll, drop, email all the students, etc.

• ArrayIntList won’t work for Courses!

• Options:
  – Create an ArrayCourseList
    • But then we have to create a custom list for every object type
  – Create an ArrayList of Objects
    • But then we have to cast when we get objects out
  – Create an ArrayList of generic objects!

Working with ArrayList

• Constructing
  \[
  \text{ArrayList}\langle\text{Course}\rangle \text{ list} = \text{new ArrayList}\langle\text{Course}\rangle();
  \]
• Adding an element to the end of the list
  \[
  \text{list}.\text{add}(\text{new Course}(\ldots));
  \]
• Adding an element at an index
  \[
  \text{list}.\text{add}(3, \text{new Course}(\ldots)); // (index, element)
  \]
• Removing an element
  \[
  \text{list}.\text{remove}(3); // (index)
  \]
• Getting an element from an index
  \[
  \text{list}.\text{get}(3); // (index)\]
ArrayList Design

- `ArrayList<E>` is a library class
  - Encapsulates helpful functionality
- Client uses the functionality to accomplish meaningful tasks
  - Tests (to evaluate library works)
  - Client programs

Testing Lists

- Test each List method
  - Empty list
  - Front of the list
  - Middle of the list
  - Back of the list
- After each operation
  - Length is correct
  - Contents are correct (are the elements in the right order?)

- Then test the methods in combination
  - maybe `add` usually works, but fails after you call `remove`
  - what happens if I call `add` then `size`? `remove` then `toString`?
  - make multiple calls; maybe `size` fails the second time only
Testing `ArrayList()`

- Start by testing the constructor
  - Should create an empty list of strings

```java
public class ArrayListTest {
  @Test
  public void testArrayList() {
    ArrayList<String> l = new ArrayList<String>();
    assertEquals(0, l.size());
  }
}
```

Why aren't we testing with `Course` objects?
If the implementation is generic, we can test with whatever object type we want!

Generics in Java

- Provide a type for elements stored in collection classes
  - Ex: `ArrayList` of Strings
- Our own classes can provide a generic type for the client to define
  - Use `E` to represent the element/type the user provides
  - `E` is a variable name for the type! And really it can be anything b/c it's a variable!
- Class name with generics
  - `ArrayIntList` to `ArrayList<E>`
  - `int` to `E` when referring to an element of the list, not the size, indexes, and capacity
### ArrayList State

- The state for a generic `ArrayList` is still an array and the size
  - Define the generic `ArrayList` class
  - Define the state for the generic `list` and `size`

<table>
<thead>
<tr>
<th>ArrayList&lt;E&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>-list: E[]</td>
</tr>
<tr>
<td>-size: int</td>
</tr>
</tbody>
</table>

- `ArrayList<E>()`
- `add(element:E):boolean`
- `add(idx:int, element:E):void`
- `remove(idx:int):E`
- `get(idx:int):E`
- `size():int`

### ArrayList Constructor

- Constructor header does NOT include the generic type!
  - Even though we include it when constructing the objects as a client.
  - The generic type is assumed from the class' header

```java
public ArrayList() {
    this(CAPACITY);
}

public ArrayList(int capacity) {
    if (capacity < 0) throw new IAE();
    //initialize list and size
}
```
Constructing an Array of Generics

- Our list is an array of a generic type, E
- list is only a reference to an array
  - An array needs to know its type and size to be constructed
  - But a generic type isn’t know until runtime!
  - So the compiler can’t understand how to create an array of an unknown type — it doesn’t know how much space to allocate
  - It also doesn’t know how to create an object of an unknown type
    - The object may not actually have a default constructor
    - Defining parameters doesn’t make the type generic!

Constructing Generic Objects

- You can’t construct objects of a generic type

```java
public class ArrayList<E> {

    private E[] list;

    public ArrayList() {
        E item = new E(); //illegal!
        list = new E[10];  //illegal!
    }
}
```
Constructing Generic Objects

• We can work with the generic type when it’s a reference
  – This includes parameters and return types
• But we need to construct something to have a list!
  – What is the most generic concrete object in Java?

Create an `Object` or `Object[]`

Cast to the generic type

Ignore the Cast warning using an `@SuppressWarnings` annotation

```java
public class ArrayList<E> {
    private E[] list;

    @SuppressWarnings("unchecked")
    public ArrayList() {
        E item = (E)(new Object());
        list = (E[])(new Object[10]);
    }
}
```
Implementing ArrayList

- Implementing ArrayList is almost exactly the same as ArrayIntList
  - Update the type for elements from int to E
  - Update primitive comparisons on elements to use equals()

```java
public int indexOf(E value) {
    for (int i = 0; i < size; i++) {
        //if (list[i] == value) {
        if (list[i].equals(value)) {
            return i;
        }
    }
    return -1;
}
```

Memory and Garbage Collection

- If you remove an element, there is still data in array locations outside the size of the array (but within the capacity)

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>3</td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>12</td>
<td>14</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>size</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Now that we’re working with objects, that means each element is a reference to an object!
- Java’s garbage collector finds objects that are no longer used (referenced) and destroys them
  - Our array list may interfere with this. The object referenced by index 5 is no longer used, but still referenced. Therefore, no garbage collection.
  - Need to explicitly set the array element back to null (removes the reference)
  - When could this be a problem?
Removing the Reference

- Set the element at index size-1 to null (before decrementing size!)

```java
public E remove(int idx) {
    checkIndex(idx);
    E v = list[idx];
    //left shift elements
    list[size - 1] = null;
    size--;
    return v;
}
```

Generic Interface

```java
public interface List<E> {
    public boolean add(E element);
    public void add(int idx, E element);
    public E get(int idx);
    public int indexOf(E value);
    public boolean isEmpty();
    public E remove(int idx);
    public E set(int idx, E value);
    public int size();
    ...
}
```

```java
public class ArrayList<E> implements List<E> { ...
```
Abstract*

- The **Abstract** classes in the Java Collections Framework provide a partial implementation of the list
- It’s still up to the user to define the list storage and size

- **Abstract** classes define behaviors that utilize other list methods
  - `add(E element)` could call `add(int idx, E element)` at `size`

AbstractList

- Intended for use for a random access data store (an array)
- Unmodifiable list
  - Implement `get(int)` & `size()`
- Modifiable list
  - Implement `set(int, E)`
- Variable-size list
  - Implement `add(int, E)` and `remove(int)`
- Provide a default constructor
- May override any other methods as needed

You’ll implement a custom list that extends AbstractList in Lab 7!
Customizing Lists

• But we have ArrayList, do we really need custom lists?
  – Yes!
• ArrayList
  – Allows null elements
  – Allows duplicates
  – Grows “forever” (until memory runs out)
  – Doesn’t maintain elements in sorted order
• Custom lists can help us create lists specific for a set of requirements

SortedList

• SortedList keeps elements in sorted order

• What is the algorithm for adding an element to a sorted list such that the list remains sorted?

• Do any other list methods need additional logic to maintain sorted order? If so, which?
SortedList and Comparable

• To compare objects, the object should implement the Comparable interface

• But what if a client tries to use a SortedList with an element type that doesn’t implement Comparable?

public class SortedList<E extends Comparable<E>> extends AbstractList<E> {
    
}
Generic Wildcards (?)

• Wildcards are used on references to generic objects when the type isn’t known
  – Represents an unknown type
  – Used as a parameter, return type, field, or local variable of a client of a generic class

• See https://docs.oracle.com/javase/tutorial/java/generics/wildcards.html for more details

Upper Bounded Wildcards

• Relax restrictions on a variable
• When used with Lists, makes the List read-only

```java
public static double sumOfList(List<? extends Number> list) {
    double sum = 0;
    for (Number n : list) {
        s += n.doubleValue();
    }
    return sum;
}
```

Works for `Integer`, `Double`, and `Number`!
Unbounded Wildcards

- Used when
  - Writing a method that can be implemented with Object functionality
  - Restricting to object functionality that doesn’t rely on the generic type
  - When working with lists, can only add null

```java
public static void printList(List<?> list) {
    for (Object e : list) {
        System.out.print(e + " ");
    }
    System.out.println();
}
```

Lower Bounded Wildcards

- Restricts the unknown type to be a specific type or a super type of that type
- Example would work with `List<Integer>`, `List<Number>`, and `List<Object>

```java
public static void addNumbers(List<? super Integer> list) {
    for (int i = 1; i <= 10; i++) {
        list.add(i);
    }
}
```