Building Java Programs

Chapter 15
Lecture 15-2: testing ArrayIntList;
pre/post conditions and exceptions

reading: 4.4 15.1 - 15.3
"Always code as if the person who ends up maintaining your code is a violent psychopath who knows where you live."

"Always code as if the person who ends up grading your code is a violent psychopath who knows where you live."

Instructor is not responsible for damages caused by TA following poor commenting, lack of boolean zen, redundant code…
Searching methods

• Implement the following methods:
  • indexOf – returns first index of element, or -1 if not found
  • contains - returns true if the list contains the given int value

• Why do we need isEmpty and contains when we already have indexOf and size?
  • Adds convenience to the client of our class:

// less elegant          // more elegant
if (myList.size() == 0) {
  if (myList.isEmpty()) {
    if (myList.indexOf(42) >= 0) {
      if (myList.contains(42)) {

Not enough space

- What to do if client needs to add more than 10 elements?

```
index 0 1 2 3 4 5 6 7 8 9
value 3 8 9 7 5 12 4 8 1 6
size 10
```

- `list.add(15); // add an 11th element`

- Possible solution: Allow the client to construct the list with a larger initial capacity.
Multiple constructors

• Our list class has the following constructor:

```java
public ArrayIntList() {
    elementData = new int[10];
    size = 0;
}
```

• Let's add a new constructor that takes a capacity parameter:

```java
public ArrayIntList(int capacity) {
    elementData = new int[capacity];
    size = 0;
}
```

• The constructors are very similar. Can we avoid redundancy?
this keyword

- **this**: A reference to the *implicit parameter* (the object on which a method/constructor is called)

- Syntax:
  - To refer to a field: \texttt{this.field}
  - To call a method: \texttt{this.method(parameters)};
  - To call a constructor from another constructor: \texttt{this(parameters)};
Revised constructors

// Constructs a list with the given capacity.
public ArrayIntList(int capacity) {
    elementData = new int[capacity];
    size = 0;
}

// Constructs a list with a default capacity of 10.
public ArrayIntList() {
    this(10);  // calls (int) constructor
}
public static final type name = value;

• **class constant**: a global, unchangeable value in a class
  • used to store and give names to important values used in code
  • documents an important value; easier to find and change later

• classes will often store constants related to that type
  • Math.PI
  • Integer.MAX_VALUE, Integer.MIN_VALUE
  • Color.GREEN

// default array length for new ArrayIntLists
public static final int DEFAULT_CAPACITY = 10;
Running out of space

- What should we do if the client starts out with a small capacity, but then adds more than that many elements?

<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>9</td>
<td>7</td>
<td>5</td>
<td>12</td>
<td>4</td>
<td>8</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>size</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `list.add(15);  // add an 11th element`

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|-------|---|---|---|---|---|---|---|---|---|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| value | 3 | 8 | 9 | 7 | 5 | 12| 4 | 8 | 1 | 6 | 15 | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |
| size  | 11|

- Answer: **Resize the array** to one twice as large.
The Arrays class

The Arrays class in `java.util` has many useful methods:

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>binarySearch(array, value)</code></td>
<td>returns the index of the given value in a <em>sorted</em> array (or &lt; 0 if not found)</td>
</tr>
<tr>
<td><code>binarySearch(array, minIndex, maxIndex, value)</code></td>
<td>returns index of given value in a <em>sorted</em> array between indexes min / max - 1 (&lt; 0 if not found)</td>
</tr>
<tr>
<td><code>copyOf(array, length)</code></td>
<td>returns a new resized copy of an array</td>
</tr>
<tr>
<td><code>equals(array1, array2)</code></td>
<td>returns true if the two arrays contain same elements in the same order</td>
</tr>
<tr>
<td><code>fill(array, value)</code></td>
<td>sets every element to the given value</td>
</tr>
<tr>
<td><code>sort(array)</code></td>
<td>arranges the elements into sorted order</td>
</tr>
<tr>
<td><code>toString(array)</code></td>
<td>returns a string representing the array, such as &quot;[10, 30, -25, 17]&quot;</td>
</tr>
</tbody>
</table>

**Syntax:** Arrays.*methodName*(parameters)
Problem: size vs. capacity

- What happens if the client tries to access an element that is past the size but within the capacity (bounds) of the array?
- Example: `list.get(7);` on a list of size 5 (capacity 10)

<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>9</td>
<td>7</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</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>

- Currently the list allows this and returns 0.
  - Is this good or bad? What (if anything) should we do about it?
Preconditions

- **precondition**: Something your method assumes is true at the start of its execution.
  - Often documented as a comment on the method's header:
    ```java
    // Returns the element at the given index.
    // Precondition: 0 <= index < size
    public int get(int index) {
        return elementData[index];
    }
    ```
  - Stating a precondition doesn't really "solve" the problem, but it at least documents our decision and warns the client what not to do.
  - What if we want to actually enforce the precondition?
Bad precondition test

- What is wrong with the following way to handle violations?

```java
// Returns the element at the given index.
// Precondition: 0 <= index < size
public int get(int index) {
    if (index < 0 || index >= size) {
        System.out.println("Bad index! " + index);
        return -1;
    }
    return elementData[index];
}
```

- returning -1 is no better than returning 0 (could be a legal value)
- `println` is not a very strong deterrent to the client (esp. GUI)
Throwing exceptions (4.4)

```java
throw new ExceptionType();
throw new ExceptionType("message");
```

- Generates an exception that will crash the program, unless it has code to handle ("catch") the exception.

- Common exception types:
  - `ArithmeticException`, `ArrayIndexOutOfBoundsException`, `FileNotFoundException`, `IllegalArgumentException`, `IllegalStateException`, `IOException`, `NoSuchElementException`, `NullPointerException`, `RuntimeException`, `UnsupportedOperationException`

- Why would anyone ever want a program to crash?
public int get(int index) {
    if (index < 0 || index >= size) {
        throw new ArrayIndexOutOfBoundsException(index);
    }
    return elementData[index];
}

• Exercise: Modify the rest of ArrayIntList to state preconditions and throw exceptions as appropriate.
Private helper methods

private type name(type name, ..., type name) {
    statement(s);
}

- a private method can be seen/called only by its own class
  - your object can call the method on itself, but clients cannot call it
  - useful for "helper" methods that clients shouldn't directly touch

private void checkIndex(int index, int min, int max) {
    if (index < min || index > max) {
        throw new IndexOutOfBoundsException(index);
    }
}
Postconditions

- **postcondition**: Something your method *promises will be true* at the *end* of its execution.
  - Often documented as a comment on the method's header:

```java
// Makes sure that this list's internal array is large enough to store the given number of elements.
// Postcondition: elementData.length >= capacity
public void ensureCapacity(int capacity) {
    // double in size until large enough
    while (capacity > elementData.length) {
        elementData = Arrays.copyOf(elementData, 2 * elementData.length);
    }
}
```

- If your method states a postcondition, clients should be able to rely on that statement being true after they call the method.
Thinking about testing

• If we wrote `ArrayIntList` and want to give it to others, we must make sure it works adequately well first.

• Some programs are written specifically to test other programs.
  • We could write a client program to test our list.
  • Its `main` method could construct several lists, add elements to them, call the various other methods, etc.
  • We could run it and look at the output to see if it is correct.

• Sometimes called a **unit test** because it checks a small unit of software (one class).
  • **black box**: Tests written without looking at the code being tested.
  • **white box**: Tests written after looking at the code being tested.
Tips for testing

- You cannot test every possible input, parameter value, etc.
  - Think of a limited set of tests likely to expose bugs.

- Think about boundary cases
  - Positive; zero; negative numbers
  - Right at the edge of an array or collection's size

- Think about empty cases and error cases
  - 0, -1, null; an empty list or array

- Test behavior in combination
  - Maybe add usually works, but fails after you call remove
  - Make multiple calls; maybe size fails the second time only
Example **ArrayIntList test**

```java
class ArrayIntList {
    int[] list;
    public ArrayIntList(int[] array) {
        list = new int[array.length];
        System.arraycopy(array, 0, list, 0, array.length);
    }
    public void add(int item) {
        if (list.length == list.capacity()) {
            list = Arrays.copyOf(list, list.capacity() + 1);
        }
        list[list.size()] = item;
    }
    public void remove(int index) {
        list[index] = list[list.size() - 1];
        list[list.size() - 1] = -1;
        list.remove(list.size() - 1);
    }
}

public static void main(String[] args) {
    int[] a1 = {5, 2, 7, 8, 4};
    int[] a2 = {2, 7, 42, 8};
    int[] a3 = {7, 42, 42};
    helper(a1, a2);
    helper(a2, a3);
    helper(new int[] {1, 2, 3, 4, 5}, new int[] {2, 3, 42, 4});
}

public static void helper(int[] elements, int[] expected) {
    ArrayIntList list = new ArrayIntList(elements);
    for (int i = 0; i < elements.length; i++) {
        list.add(elements[i]);
    }
    list.remove(0);
    list.remove(list.size() - 1);
    list.add(2, 42);
    for (int i = 0; i < expected.length; i++) {
        if (list.get(i) != expected[i]) {
            System.out.println("fail; expect " + Arrays.toString(expected) + ", actual " + list);
        }
    }
}
```