

Building Java Programs

Chapter 15

testing `ArrayList`;
pre/post conditions and exceptions

reading: 4.4 15.1 - 15.3



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

```
if (myList.size() == 0) {  
if (myList.indexOf(42) >= 0) {
```

// more elegant

```
if (myList.isEmpty()) {  
if (myList.contains(42)) {
```

Not enough space

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

| | | | | | | | | | | |
|--------------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| <i>index</i> | <i>0</i> | <i>1</i> | <i>2</i> | <i>3</i> | <i>4</i> | <i>5</i> | <i>6</i> | <i>7</i> | <i>8</i> | <i>9</i> |
| <i>value</i> | 3 | 8 | 9 | 7 | 5 | 12 | 4 | 8 | 1 | 6 |
| <i>size</i> | 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:

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

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

```
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: `this.field`
 - To call a method: `this.method (parameters) ;`
 - To call a constructor from another constructor: `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  
}
```

Class constants

```
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?

| | | | | | | | | | | |
|--------------|----|---|---|---|---|----|---|---|---|---|
| <i>index</i> | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| <i>value</i> | 3 | 8 | 9 | 7 | 5 | 12 | 4 | 8 | 1 | 6 |
| <i>size</i> | 10 | | | | | | | | | |

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

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

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

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)

| | | | | | | | | | | |
|--------------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| <i>index</i> | <i>0</i> | <i>1</i> | <i>2</i> | <i>3</i> | <i>4</i> | <i>5</i> | <i>6</i> | <i>7</i> | <i>8</i> | <i>9</i> |
| <i>value</i> | 3 | 8 | 9 | 7 | 5 | 0 | 0 | 0 | 0 | 0 |
| <i>size</i> | 5 | | | | | | | | | |

- 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:

```
// 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?

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

Throwing exceptions (4.4)

```
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?

Exception example

```
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);  
    }  
}
```

The Arrays class

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

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

- Syntax: `Arrays.methodName(parameters)`

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:

```
// 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
    int newCapacity = elementData.length;
    while (capacity > newCapacity) {
        newCapacity = newCapacity * 2;
    }
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

- 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

```
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);
        }
    }
}
}
```