CSE 143

Lecture 3: ArrayIntList;
pre/post conditions and exceptions
reading: 4.4 15.1 - 15.3
Why does style matter?

- Maintenance
  - // magic number
    int magicNumber = 9;

- Getting a job
  - Every company has a different style guide
Implementing \texttt{remove}

• Again, we need to shift elements in the array
  • this time, it's a left-shift
  • in what order should we process the elements?
  • what indexes should we process?

\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|}
  \hline
  \textbf{index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
  \hline
  \textbf{value} & 3 & 8 & 9 & 7 & 5 & 12 & 0 & 0 & 0 & 0 \\
  \hline
  \textbf{size} & 6 & & & & & & & & & \\
  \hline
\end{tabular}

• \texttt{list.remove(2);}  // delete 9 from index 2

\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|}
  \hline
  \textbf{index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
  \hline
  \textbf{value} & 3 & 8 & 7 & 5 & 12 & 0 & 0 & 0 & 0 & 0 \\
  \hline
  \textbf{size} & 5 & & & & & & & & & \\
  \hline
\end{tabular}
public void remove(int index) {
    for (int i = index; i < size; i++) {
        list[i] = list[i + 1];
    }
    size--;
    list[size] = 0; // optional (why?)
}

list.remove(2); // delete 9 from index 2

<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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>size</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 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)

```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?
Exception example

```java
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.
Not enough space

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

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

- Possible solution: Allow the client to construct the list with a larger initial 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>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>
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:  
  ```java
  this.field
  ```
- To call a method:  
  ```java
  this.method(parameters);
  ```
- To call a constructor from another constructor:  
  ```java
  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  | 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 <code>min/max</code> - 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 <code>true</code> 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>
</tbody>
</table>
| `toString(array)`                               | returns a string representing the array, such as 

```
[10, 30, -25, 17]
```

- **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?
Private helper methods

```java
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

```java
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.