CSE 143

Lecture 2: Implementing `ArrayIntList`

reading: 15.1 - 15.3
Wrapper classes

<table>
<thead>
<tr>
<th>Primitive Type</th>
<th>Wrapper Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
<tr>
<td>char</td>
<td>Character</td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

- A **wrapper** is an object whose sole purpose is to hold a primitive value.

- Once you construct the list, use it with primitives as normal:

```java
ArrayList<Double> grades = new ArrayList<Double>();
grades.add(3.2);
grades.add(2.7);
...
double myGrade = grades.get(0);
```
Recall: classes and objects

- **class**: A program entity that represents:
  - A complete program or module, or
  - A template for a type of objects.
  - *ArrayList* is a class that defines a type.

- **object**: An entity that combines state and behavior.
  - **object-oriented programming (OOP)**: Programs that perform their behavior as interactions between objects.
  - **abstraction**: Separation between concepts and details. Objects provide abstraction in programming.
Elements of a class

public class BankAccount {
    private String name;       // fields:
    private int id;            // data encapsulated
    private double balance;    // inside each object

    public BankAccount(String name, int id) {
        this.name = name;         // constructor:
        this.id = id;              // initializes
        this.balance = 0.0;        // new objects
    }

    public void deposit(double amount) {
        this.balance += amount;   // instance method:
    }

    // each object's behavior
}

"implicit parameter": object on which a method was called
**ArrayList implementation**

- What is an `ArrayList`'s behavior?
  - add, remove, `indexOf`, etc

- What is an `ArrayList`'s state?
  - Many elements of the same type
  - For example, unfilled array

<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>...</th>
<th>98</th>
<th>99</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>17</td>
<td>932085</td>
<td>-32053278</td>
<td>100</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>...</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>
ArrayIntList implementation

- Simpler than ArrayList\<E\>
  - No generics (only stores ints)
  - Fewer methods: add(value), add(index, value), get(index), set(index, value), size(), isEmpty(), remove(index), indexOf(value), contains(value), toString(),

- Fields?
  - int[]
  - int to keep track of the number of elements added
  - The default capacity (array length) will be 10
Printing an `ArrayIntList`

- Let's add a method that allows clients to print a list's elements.
  - You may be tempted to write a `print` method:

    ```
    // client code
    ArrayIntList list = new ArrayIntList();
    ...
    list.print();
    ```

  - Why is this a bad idea? What would be better?
The `toString` method

- Tells Java how to convert an object into a `String`
  ```java
  ArrayIntList list = new ArrayIntList();
  System.out.println("list is " + list);
  // ("list is " + list.toString());
  ```

- Syntax:
  ```java
  public String toString() {
      code that returns a suitable String;
  }
  ```

- Every class has a `toString`, even if it isn't in your code.
  - The default is the class's name and a hex (base-16) number:
    ```java
    ArrayIntList@9e8c34
    ```
toString solution

// Returns a String representation of the list.
public String toString() {
    if (size == 0) {
        return "[]";
    } else {
        String result = "[" + elementData[0];
        for (int i = 1; i < size; i++) {
            result += ", " + elementData[i];
        }
        result += "]";
        return result;
    }
}
Implementing \texttt{add #2}

- How do we add to the middle or end of the list?
  - must \textit{shift} elements to make room for the value (see book 7.4)

<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>

- \texttt{list.add(3, 42);} // insert 42 at index 3

<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>42</td>
<td>7</td>
<td>5</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>size</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Note: The order in which you traverse the array matters!
add #2 code

public void add(int index, int value) {
    for (int i = size; i > index; i--) {
        list[i] = list[i - 1];
    }
    list[index] = value;
    size++;
}

- list.add(3, 42);

<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>

<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>42</td>
<td>7</td>
<td>5</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>size</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Other methods

- Let's implement the following methods in our list:
  
  - **get(index)**
    Returns the element value at a given index.
  
  - **set(index, value)**
    Sets the list to store the given value at the given index.
  
  - **size()**
    Returns the number of elements in the list.
  
  - **isEmpty()**
    Returns `true` if the list contains no elements; else `false`.
    (Why write this if we already have the `size` method?)
Implementing `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?

```java
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>

<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>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>
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