CSE 143X

Accelerated Computer Programming I/II
ArrayIntList

I made a JavaList!

[Espresso, cappuccino, mocha]
Wrapper Classes

int vs. Integer  char vs. Character  double vs. Double

The lowercase versions are **primitive types**; the uppercase versions are “wrapper classes”.

The following is valid code:

```java
1 int a = 5;
2 Integer b = 10;
3 int c = a + b; // You can treat ints and Integers as the same
```

When we create ArrayList’s, we must use **non-primitive types**. So:

```java
1 ArrayList<int> bad1 = new ArrayList<int>(); // This won’t compile!
2 // v This will work.
3 ArrayList<Integer> better = new ArrayList<Integer>();
4 better.add(5); // We can add an 'int' to an 'Integer' ArrayList
```
Client vs. Implementor: Medication

For a tylenol pill, who is the client? Who is the implementor?

Java Examples

You’ve already been a client!

- DrawingPanel
- ArrayList

You’ve already been an implementor!

- Critter
Class

A Class is

- a complete program, or
- a “template” for a type

(Examples: ArrayList, ReverseFile, ...)

The class explains what an object is, an instance is a particular version of the object.

```
1  ArrayList<String> list1 = new ArrayList<String>();
2  ArrayList<String> list2 = new ArrayList<String>();
3  //list1 and list2 are instances of ArrayList
```

Object

An Object combines state and behavior.

Java is an “object-oriented” programming language (OOP); programs consist of objects interacting with each other.
A class is made up of **field(s)**, **constructor(s)**, and **method(s)**. Let’s make an object Circle that represents a circle...

- with a size
- that can be moved right
- at a particular location

```java
public class Circle {
    /* Fields */
    private int radius;
    private int x;
    private int y;

    /* Constructor */
    public Circle(int radius, int x, int y) {
        this.radius = radius;
        this.x = x;
        this.y = y;
    }

    /* Methods */
    public void moveRight(int numberOfUnits) {
        this.x += numberOfUnits;
    }
}
```
Implementor View of ArrayList

What behavior should we support? (Methods)
- add, remove, indexOf, etc.

What state do we keep track of? (Fields)
- Elements stored in the ArrayList (probably stored as an array!)
- Size of ArrayList

Two Views of an ArrayList

<table>
<thead>
<tr>
<th>Client View:</th>
<th>3</th>
<th>-23</th>
<th>-5</th>
<th>222</th>
<th>35</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Impl. View:</th>
<th>3</th>
<th>-23</th>
<th>-5</th>
<th>222</th>
<th>35</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>
- 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()
Implementing `add` function:

1. **lst.add(222):**

   - **(size = 5)**

How do we add to the end of the list?

- Put the element in the last slot
- Increment the size

```java
public void add(int value) {
    this.data[this.size] = value;
    this.size++;
}
```
System.out.println automatically calls toString on the given object. toString looks like:

```java
public String toString() {
  ...
}
```

ArrayIntList toString:

```java
public String toString() {
  if (this.size == 0) {
    return "[]";
  } else {
    String result = "[" + this.data[0];
    for (int i = 1; i < this.size; i++) {
      result += ", " + this.data[i];
    }
    result += "]";
    return result;
  }
}
```
### Implementing add #2

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>8</td>
<td>2</td>
<td>45</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

```
list.add(1, 222):
```

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>222</td>
<td>8</td>
<td>2</td>
<td>45</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- How do we add to the middle of the list?
  - Shift over all elements starting from the end
  - Put the new element in its index
  - Increment the size

```java
public void add(int index, int value) {
    for (int i = this.size; i > index; i--) {
        this.data[i] = this.data[i - 1];
    }
    this.data[index] = value;
    this.size++;
}
```
Today’s Takeaways!

- Understand the difference between client and implementor
- Always use wrapper classes when creating an ArrayList of a primitive type
- Understand how ArrayList is implemented
More ArrayIntList;
pre/post; exceptions;
debugging

Sometimes, you just have to go backwards.
What Are We Doing Again?

What Are We Doing...?
We’re implementing our own (simpler) version of ArrayList to (a) see how it works, and (b) get experience being the “implementor” of a class.

And how does the client see all of our comments...?

Today’s Main Goal:
To finish ArrayIntList!
Outline

1. Debugging
2. More Functionality
3. Removing Code Duplication
4. Improving Readability!
5. Preventing Malicious Behavior
6. Re-structuring the Code
WTF’s per Minute

Code Quality Measurement: WTFs/Minute

Good Code

Bad Code

http://commadot.com
What is this code supposed to do? What does it do?

```java
public class WTF {
    public static void main(String[] args) {
        ArrayIntList list1 = new ArrayIntList();
        ArrayIntList list2 = new ArrayIntList();
        list1.add(5);
        list2.add(5);
        if (list1 == list2) {
            System.out.println("Yay!");
        }
        else {
            System.out.println("Boo.");
        }
    }
}
```

Rubber Ducky Debugging

Rubber Ducky Debugging is the idea that when your code doesn’t work, you talk to an inanimate object about what it does to find the error.

The idea is to verbalize what your code is supposed to do vs. what it is doing. Just saying it out loud helps solve the problem.
## Implementing `remove`

- Original list: \(\text{size} = 5\) 3 8 2 45 6 0 0 0

\[
\begin{array}{cccccccc}
\text{list[0]} & \text{list[1]} & \text{list[2]} & \text{list[3]} & \text{list[4]} & \text{list[5]} & \text{list[6]} & \text{list[7]} \\
3 & 8 & 2 & 45 & 6 & 0 & 0 & 0 \\
\end{array}
\]

- After `list.remove(2)`:

\[
\begin{array}{cccccccc}
\text{list[0]} & \text{list[1]} & \text{list[2]} & \text{list[3]} & \text{list[4]} & \text{list[5]} & \text{list[6]} & \text{list[7]} \\
3 & 8 & 45 & 6 & 0 & 0 & 0 & 0 \\
\end{array}
\]

### How do we remove from the middle of the list?

- Shift over all elements starting from the index to remove at
- Set the last element to 0 (Do we **need** to do this?)
- Decrement the size
The fix is to call the **more general** add method from the **less general** one. (As a rule of thumb, methods with fewer arguments are less general.) So, we’d replace the **first** method with:

**Fixed add Method**

```java
public void add(int value) {
    add(this.size, value);
}
```
We'd like to have two constructors for ArrayIntList:
- One that uses a default size
- One that uses a size given by the user

This is a lot of redundant code! How can we fix it?

Java allows us to call one constructor from another using this(...):

```java
public ArrayIntList() {
    this(10);
}
```
Looking back at the constructor, what's ugly about it?

```java
public ArrayIntList() {
    this(10);
}
```

The 10 is a “magic constant”; this is really bad style!! We can use:

```java
public static final type name = value
```

to declare a **class constant**.

So, for instance:

```java
public static final int DEFAULT_CAPACITY = 10.
```

---

A class constant is a **global, unchangable** value in a class. Some examples:

- Math.PI
- Integer.MAX_VALUE, Integer.MIN_VALUE
- Color.GREEN
Are there any arguments to `moveRight` that are “invalid”?

**Yes!** We shouldn’t allow negative numbers.

The implementor is responsible for (1) telling the user about invalid ways to use methods and (2) preventing a malicious user from getting away with using their methods in an invalid way!
A **precondition** is an assertion that something must be true for a method to work correctly. The objective is to tell clients about invalid ways to use your method.

Example Preconditions:

- For `moveRight(int numberOfUnits)`: 

- For `minElement(int[] array)`: 

- For `add(int index, int value)`: 

Preconditions are important, because they explain method behavior to the client, but they aren’t enough! The client can still use the method in invalid ways!
Preconditions

A **precondition** is an assertion that something must be true for a method to work correctly. The objective is to tell clients about invalid ways to use your method.

**Example Preconditions:**

- For `moveRight(int numberOfUnits)`:  
  ```java
  // pre: numberOfUnits >= 0
  ```

- For `minElement(int[] array)`:  
  ```java
  // pre: array.length > 0
  ```

- For `add(int index, int value)`:  
  ```java
  // pre: capacity >= size + 1; 0 <= index <= size
  ```

Preconditions are important, because they explain method behavior to the client, but **they aren’t enough**! The client can still use the method in invalid ways!
An exception is an indication to the programmer that something unexpected has happened. When an exception happens, the program immediately stops running.

To make an exception happen:

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

Common Exception Types

- ArithmeticException
- ArrayIndexOutOfBoundsException
- FileNotFoundException
- IllegalArgumentException
- IllegalStateException
- IOException
- NoSuchElementException
- NullPointerException
- RuntimeException
- UnsupportedOperationException
- IndexOutOfBoundsException
**Why Use Exceptions?**

Exceptions prevent the client from accidentally using the method in a way it wasn’t intended. They alert them about errors in their code!

**An Example**

```java
public void set(int index, int value) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException(index);
    }
    this.data[index] = value;
}

public int get(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException(index);
    }
    return data[index];
}
```

Uh oh! We have MORE redundant code!
A **private method** is a method that **only the implementor** can use. They are useful to abstract out redundant functionality.

**Better set/get**

```java
private void checkIndex(int index, int max) {
    if (index < 0 || index > max) {
        throw new IndexOutOfBoundsException(index);
    }
}

d public void set(int index, int value) {
    checkIndex(index, size - 1);
    this.data[index] = value;
}

d public int get(int index) {
    checkIndex(index, size - 1);
    return data[index];
}
Let’s run `add(3, 8)`! Uh oh! There’s no space left. What do we do?

Create a new array of `double` the size, and copy the elements!
**Arrays Reference**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>binarySearch(array, val)</code></td>
<td>Returns the index of <code>val</code> in <code>array</code> if <code>array</code> is sorted; (or &lt; 0 if not found)</td>
</tr>
<tr>
<td><code>toString()</code></td>
<td>Returns a string representation of the array such as <code>[3, 42, -7, 15]</code></td>
</tr>
<tr>
<td><code>sort(array)</code></td>
<td>Sorts the elements of <code>array</code> (this edits the original array!)</td>
</tr>
<tr>
<td><code>copyOf(array, len)</code></td>
<td>Returns a new copy of <code>array</code> with length <code>len</code></td>
</tr>
<tr>
<td><code>equals(array1, array2)</code></td>
<td>Returns true precisely when the elements of <code>array1</code> and <code>array2</code> are identical (according to <code>.equals</code>)</td>
</tr>
</tbody>
</table>

Call these with `Arrays.method(arg1, arg2, ...)`
Postconditions

A postcondition is an assertion that something must be true after a method has run. The objective is to tell clients what your method does.

Example Postconditions:

- For `moveRight(int numberOfUnits)`: // post: Increases the x coordinate of the circle by `numberOfUnits`

- For `minElement(int[] array)`: // post: returns the smallest element in array

- For `add(int index, int value)`: // post: Inserts value at index in the ArrayList; shifts all elements from index to the end forward one index; ensures capacity of ArrayList is large enough

Postconditions are important, because they explain method behavior to the client.
A postcondition is an assertion that something must be true after a method has run. The objective is to tell clients what your method does.

Example Postconditions:

- For moveRight(int numberOfUnits):
  // post: Increases the x coordinate of the circle by numberOfUnits

- For minElement(int[] array):
  // post: returns the smallest element in array

- For add(int index, int value):
  // post: Inserts value at index in the ArrayList;
  // shifts all elements from index to the end forward one index; ensures capacity of
  // ArrayList is large enough

Postconditions are important, because they explain method behavior to the client.