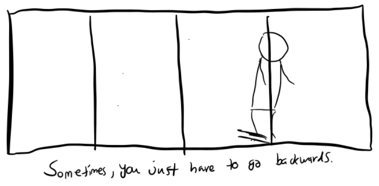
Adam Blank | Lecture 3 | Spring 2015

CSE 143

Computer Programming II

CSE 143: Computer Programming II

More `ArrayIntList`; pre/post; exceptions; debugging



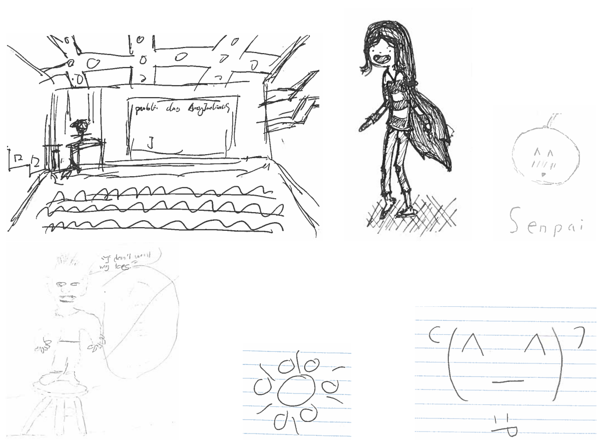
Questions From Last Time

- Mac or windows? (I usually use a Mac.)
- Why don't you like emacs? (Because I learned vim first.)
- If you go to a conference and are given a "Hello my name is" name tag, what do you write? (Sometimes I draw a line.)
- Where are you from? (Somewhere worse than Seattle.)
- Something about shark week? (Ionno man.)
- What is your opinion of netbeans? (If you're going to use an IDE, almost everyone uses Eclipse; so, I'd use Eclipse.)
- Will we be told when we need to check for bad input or throw exceptions? (Yes.)
- Will old exams be posted? (Yes.)
- Can you tell us a little bit about yourself please? (Yes, but what?)
- How do you turn in the homework? (See the turn-in button. You will turn in your Java files.)

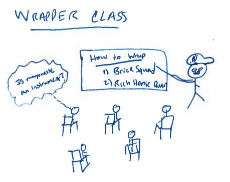
Drawings

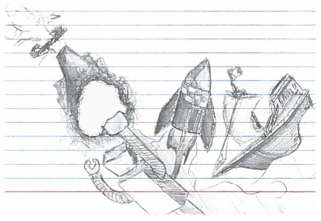
Drawings 2

Drawings 3



Drawings 4





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.

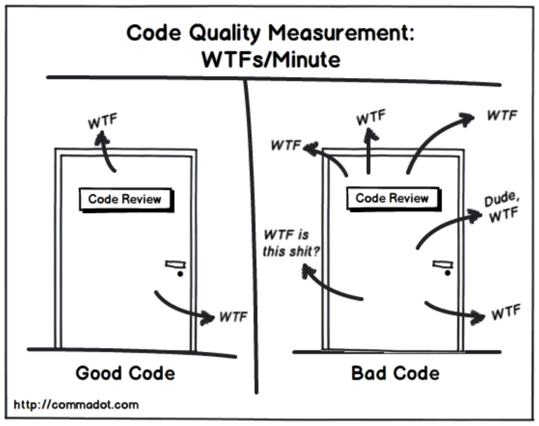
And how does the client see all of our comments...?

BTW, I did some of the boring code for us...

Today's Main Goal:
To finish `ArrayIntList`!

Outline

WTF's per Minute



Rubber Ducky, You're The One!

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 Duck Debugging

Rubber Duck 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`

(size = 5)

3	8	2	45	6	0	0	0
list[0]	list[1]	list[2]	list[3]	list[4]	list[5]	list[6]	list[7]

`list.remove(2)`:
(size = 3)

3	8	**45**	**6**	0	0	0	0
list[0]	list[1]	list[2]	list[3]	list[4]	list[5]	list[6]	list[7]

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

Duplicated Code: Methods

Redundant add Methods

```java
/* Inside the ArrayIntList class... */
public void add(int value) {
   this.set(size, value); /* THIS LINE IS DUPLICATED BELOW!!! */
   this.size++;           /* THIS LINE IS DUPLICATED BELOW!!! */
}

/* Inserts value into the list at index. */
public void add(int index, int value) {
   for (int i = size; i > index; i--) {
      this.set(i, this.get(i-1));
   }
   this.set(size, value); /* THIS LINE IS DUPLICATED ABOVE!!! */
   this.size++;           /* THIS LINE IS DUPLICATED ABOVE!!! */
}
```

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);
}
```

Duplicated Code: Constructors

We'd like to have two constructors for `ArrayIntList`:

- One that uses a default size
- One that uses a size given by the user

Redundant Constructors

```java
/* Inside the ArrayIntList class... */
public ArrayIntList() {
   this.data = new int[10];
   this.size = 0;
}

public ArrayIntList(int capacity) {
   this.data = new int[capacity];
   this.size = 0;
}
```

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

Fixed Constructor

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

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

Class CONSTANTS

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:

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

to declare a **class constant**.

So, for instance:

`public static final int DEFAULT_CAPACITY = 10.`

Class CONSTANT

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

- `Math.PI`
- `Integer.MAX_VALUE`, `Integer.MIN_VALUE`
- `Color.GREEN`

Illegal Arguments

```java
public class Circle {
   int radius;
   int x, y;
   ...

   public void moveRight(int numberOfUnits) {
      this.x += numberOfUnits;
   }
}
```

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!

Preconditions

Precondition

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)`:

  `// pre: numberOfUnits >= 0`

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

  `// pre: array.length > 0`

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

  `// 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!

Exceptions

Exceptions

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!

Private Methods

Private Methods

A **private method** is a method that **only the implementor** can use. They are useful to abstract out redundant functionality.

Better set/get

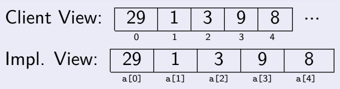
```java
private void checkIndex(int index, int max) {
    if (index < 0 || index > max) {
        throw new IndexOutOfBoundsException(index);
    }
}

public void set(int index, int value) {
    checkIndex(0, size - 1);
    this.data[index] = value;
}

public int get(int index) {
    checkIndex(0, size - 1);
    return data[index];
}
```

Hitting Capacity

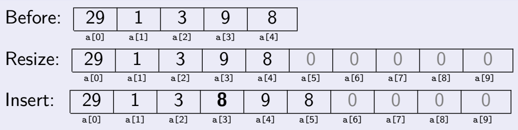
Example ArrayList



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!

Resizing (Implementor View)



Before:
29 1 3 9 8
a[0] a[1] a[2] a[3] a[4]
Resize:
29 1 3 9 8 0 0 0 0 0
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]
Insert:
29 1 3 8 9 8 0 0 0 0
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]

Arrays Reference

`binarySearch(`**array**`, `**val**`)`	Returns the index of **val** in **array** **if array is sorted**; (or <0 if not found)
`toString()`	Returns a string representation of the array such as `[3, 42, -7, 15]`
`sort(`**array**`)`	Sorts the elements of **array** (this edits the original array!)
`copyOf(`**array**`, `**len**`)`	Returns a **new** copy of **array** with length **len**
`equals(`**array1**`, `**array2**`)`	Returns true precisely when the elements of **array1** and **array2** are identical (according to `.equals`)

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

Postconditions

Postcondition

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.