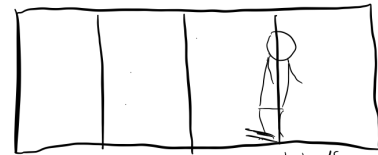


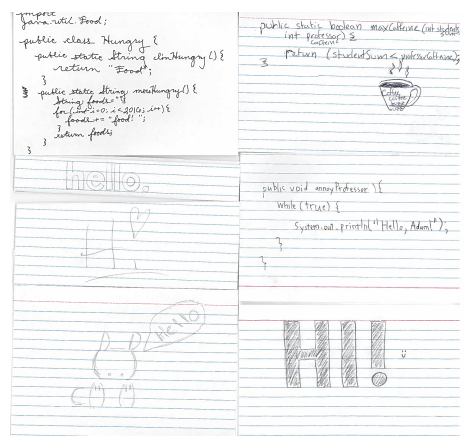
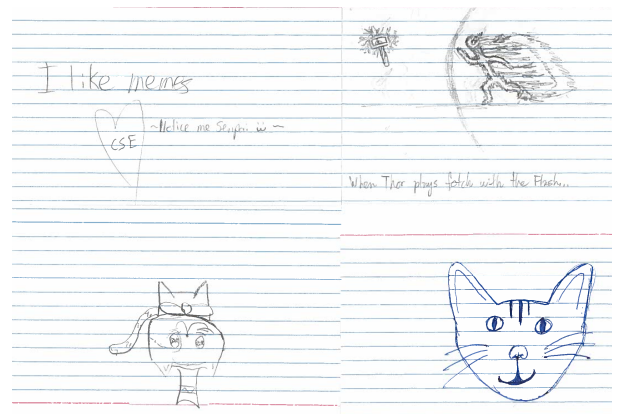
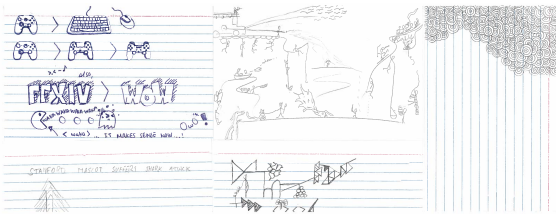
CSE 143

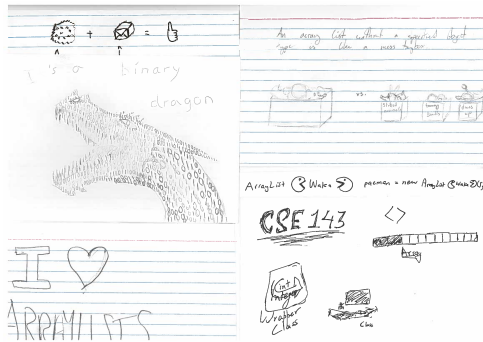
Computer Programming II

More ArrayList; pre/post; exceptions; debugging



Sometimes, you just have to go backwards.





- Eclipse is not installing: (1) make sure you are downloading Neon, (2) uninstall and reinstall the JDK, (3) come to office hours/IPL
- Is Viz open source? Not yet, but it will be.
- `Integer.toString(i)` or `i + ""`; either is fine
- You should always use "this" when you can—see the style guidelines
- Can we use Javadoc (e.g., "@params")? Yes.
- Are there any bugs that have shipped in Java. Yes; we'll talk about some later.
- Do I have to put my curly brace on the same line? No, but be consistent.

- What's the difference between primitives and Objects? Primitives are built into Java and handled differently internally. You really don't have to know the difference, but I can explain in depth at office hours.
- Mac or PC? Doesn't matter!
- Use Java 1.8
- How do I do well in this class? Do LOTS of practice-it problems.
- When will we have everything we need for homework? It varies week-by-week. You can almost always start the HW the day it comes out; we may go through more useful things throughout the week, but they shouldn't block you from making progress.

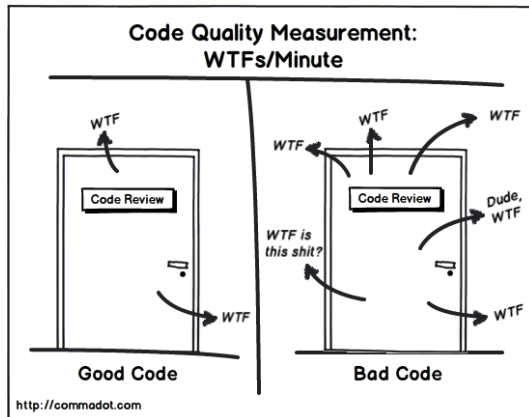
- List vs. ArrayList
- "this"
- What exactly happens when ArrayList copies the array to get more space?
- How does ArrayList know how many more spaces to put?
- What's wrong with the `toString()` method?

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.

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

- 1 Debugging
- 2 Removing Code Duplication
- 3 Improving Readability!
- 4 Preventing Malicious Behavior
- 5 Re-structuring the Code
- 6 Extras



A Bug?

```

1 public String toString() {
2     String result = "[";
3     for (int i = 0; i < this.size() - 1; i++) {
4         result += this.elementData[i] + ", ";
5     }
6
7     result += this.elementData[this.size() - 1];
8     result += "]";
9
10    return result;
11 }

```

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.

In add, we run the following line:

```
this.elementData[size] = element;
```

By default, unless we use the new keyword, all arrays are null. null is Java's way of saying "you haven't told me what to put here yet". So, if this.elementData is null, then what is null[size]?

The take-away here is that finding an "index" of null doesn't make sense. We should ensure that we assign this.elementData to be a new array if we want to use it.

Example ArrayList

Client View:

29	1	3	9	8	...
0	1	2	3	4	

Impl. View:

29	1	3	9	8
a[0]	a[1]	a[2]	a[3]	a[4]

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]

Redundant add Methods

```

1 /* Inside the ArrayList class... */
2 public void add(int value) {
3     this.set(size, value); /* THIS LINE IS DUPLICATED BELOW!!! */
4     this.size++;          /* THIS LINE IS DUPLICATED BELOW!!! */
5 }
6
7 /* Inserts value into the list at index. */
8 public void add(int index, int value) {
9     for (int i = size; i > index; i--) {
10        this.set(i, this.get(i-1));
11    }
12    this.set(size, value); /* THIS LINE IS DUPLICATED ABOVE!!! */
13    this.size++;          /* THIS LINE IS DUPLICATED ABOVE!!! */
14 }

```

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

```

1 public void add(int value) {
2     add(this.size, value);
3 }

```

We'd like to have two constructors for ArrayList:

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

Redundant Constructors

```

1 /* Inside the ArrayList class... */
2 public ArrayList() {
3     this.data = new int[10];
4     this.size = 0;
5 }
6
7 public ArrayList(int capacity) {
8     this.data = new int[capacity];
9     this.size = 0;
10 }

```

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(...):

```

1 public ArrayList() {
2     this(10);
3 }

```

Looking back at the constructor, what's ugly about it?

```
1 public ArrayIntList() {
2     this(10);
3 }
```

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, unchangeable** value in a class. Some examples:

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

```
1 public class Circle {
2     int radius;
3     int x, y;
4     ...
5
6     public void moveRight(int numberOfUnits) {
7         this.x += numberOfUnits;
8     }
9 }
```

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!

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

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

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

```
1 public void set(int index, int value) {
2     if (index < 0 || index >= size) {
3         throw new IndexOutOfBoundsException(index);
4     }
5     this.data[index] = value;
6 }
7
8 public int get(int index) {
9     if (index < 0 || index >= size) {
10        throw new IndexOutOfBoundsException(index);
11    }
12    return data[index];
13 }
```

Uh oh! We have MORE redundant code!

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

```
1 private void checkIndex(int index, int max) {
2     if (index < 0 || index > max) {
3         throw new IndexOutOfBoundsException(index);
4     }
5 }
6
7 public void set(int index, int value) {
8     checkIndex(index, size - 1);
9     this.data[index] = value;
10 }
11
12 public int get(int index) {
13     checkIndex(index, size - 1);
14     return data[index];
15 }
```

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.

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

```

1 public class Circle {
2     int radius;
3     int x, y;
4
5     public Circle(int radius, int x, int y) {
6         this.radius = radius;
7         this.x = x;
8         this.y = y;
9     }
10
11    public void setX(int x) {
12        this.x = x;
13    }
14    public int getX() {
15        return this.x;
16    }
17
18    /* There are two possible implementations of moveRight... */
19    public void moveRight(int numberOfUnits) {
20        this.x += numberOfUnits;
21    }
22    public void moveRight(int numberOfUnits) {
23        this.setX(this.getX() + numberOfUnits);
24    }
25 }

```

Why Use Fields Directly?

- It's sometimes more readable to use the fields directly
- The code is sometimes shorter

Why Use Getters and Setters?

- What happens if we change the implementation (e.g. Point location instead of `int x, y`)?
- If there is code that checks validity of inputs, then we only put it in one place

(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