CSE 143X

Accelerated Computer Programming I/II

ArrayIntList

`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:

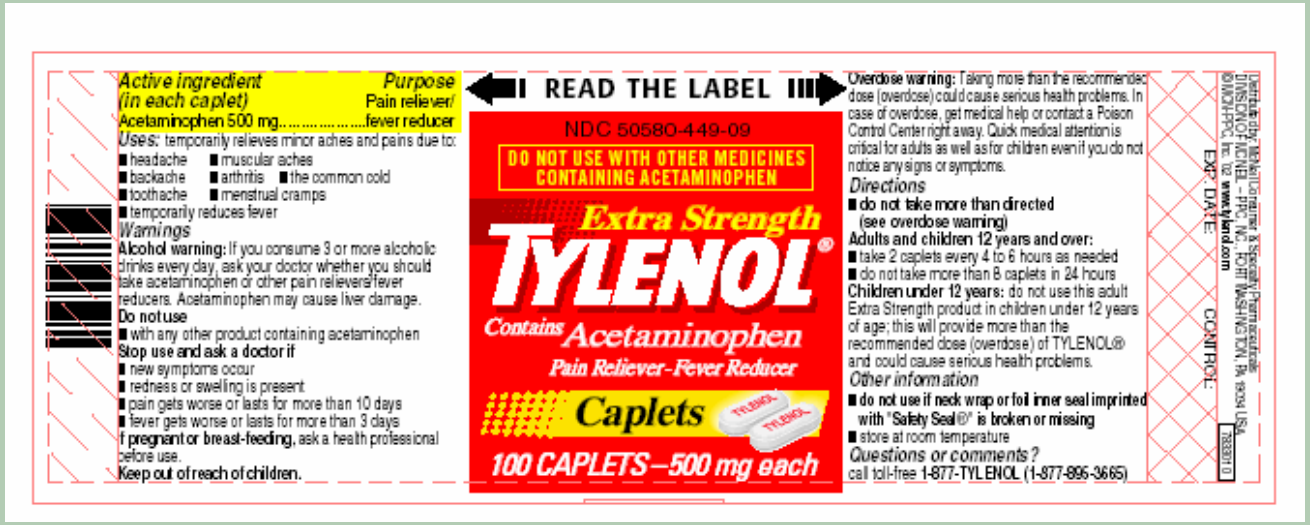
```
1 int a = 5;
2 Integer b = 10;
3 int c = a + b; //You can treat ints and Integers as the same
```

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

```
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

```
1  public class Circle {
2     /* Fields */
3     private int radius;
4     private int x;
5     private int y;
6
7     /* Constructor */
8     public Circle(int radius, int x, int y) {
9        this.radius = radius;
10       this.x = x;
11       this.y = y;
12    }
13
14    /* Methods */
15    public void moveRight(int numberOfUnits) {
16       this.x += numberOfUnits;
17    }
18 }
```

Implementor View of `ArrayList`

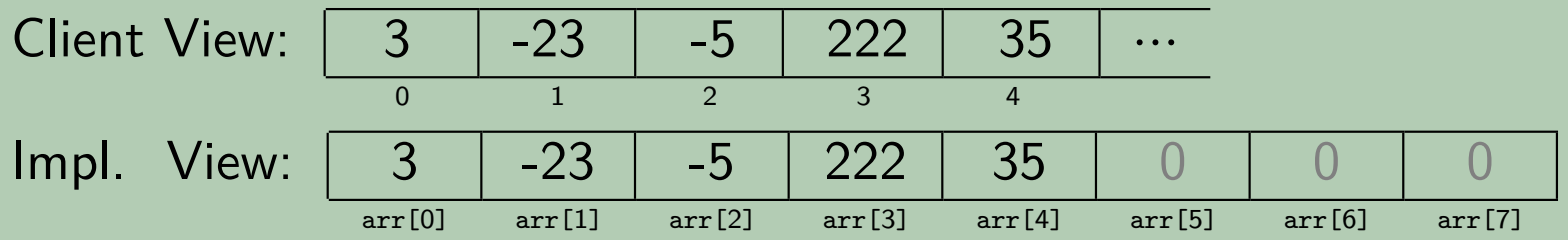
What behavior should we support? (Methods)

`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`



- 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`

(size = 4)

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

`lst.add(222)`:

(size = 5)

3	8	2	45	**222**	0	0	0
`lst[0]`	`lst[1]`	`lst[2]`	`lst[3]`	`lst[4]`	`lst[5]`	`lst[6]`	`lst[7]`

How do we add to the end of the list?

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

```
1 public void add(int value) {
2    this.data[this.size] = value;
3    this.size++;
4 }
```

Printing an ArrayIntList

`System.out.println` automatically calls `toString` on the given object. `toString` looks like:

```
1  public String toString() {
2     ...
3  }
```

`ArrayIntList toString`:

```
1  public String toString() {
2     if (this.size == 0) {
3        return "[]";
4     }
5     else {
6        String result = "[" + this.data[0];
7        for (int i = 1; i < this.size; i++) {
8           result += ", " + this.data[i];
9        }
10       result += "]";
11       return result;
12    }
13 }
```

Implementing `add` #2

(size = 4)

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

`list.add(1, 222)`:

(size = 5)

3	**222**	**8**	**2**	**45**	0	0	0
list[0]	list[1]	list[2]	list[3]	list[4]	list[5]	list[6]	list[7]

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

```
1 public void add(int index, int value) {
2    for (int i = this.size; i > index; i--) {
3       this.data[i] = this.data[i - 1];
4    }
5    this.data[index] = value;
6    this.size++;
7 }
```

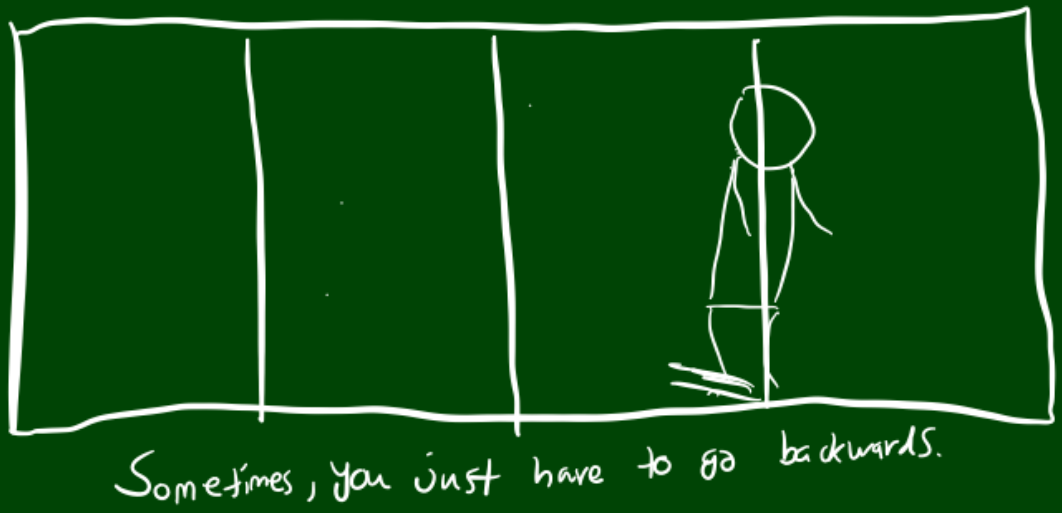
Today's Takeaways!

- Understand the difference between client and implementor
- Always use wrapper classes when creating an `ArrayList` of a primitive type
- Understand how `ArrayList` is implemented

CSE 143X

Accelerated Computer Programming I/II

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



What Are We Doing Again?

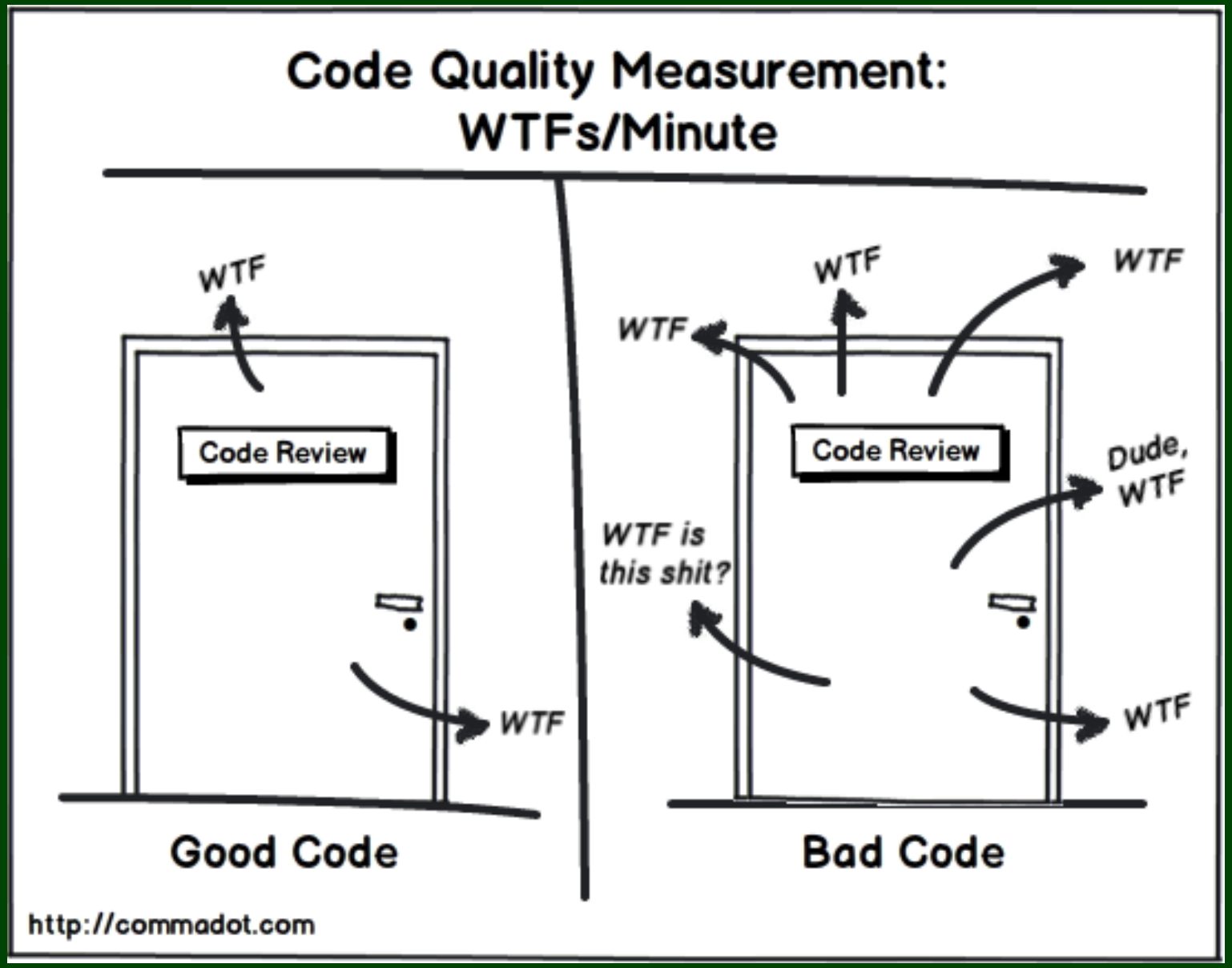
What Are We Doing. . . ?

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


Code Quality Measurement:
WTFs/Minute
WTF
Code Review
WTF
Good Code
WTF
WTF
WTF
WTF
Code Review
Dude,
WTF
WTF is
this shit?
WTF
Bad Code
http://commadot.com

What is this code supposed to do? What does it do?

```
1  public class WTF {
2     public static void main(String[] args) {
3        ArrayIntList list1 = new ArrayIntList();
4        ArrayIntList list2 = new ArrayIntList();
5        list1.add(5);
6        list2.add(5);
7        if (list1 == list2) {
8           System.out.println("Yay!");
9        }
10       else {
11          System.out.println("Boo.");
12       }
13    }
14 }
```

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

Redundant add Methods

```
1  /* Inside the ArrayIntList 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 `ArrayIntList`:

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

Redundant Constructors

```
1  /* Inside the ArrayIntList class... */
2  public ArrayIntList() {
3     this.data = new int[10];
4     this.size = 0;
5  }
6
7  public ArrayIntList(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 ArrayIntList() {
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**, **unchangable** 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)`:
- 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!

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`

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

```
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 }
```

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]

`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, ...)`

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)`:
- For `minElement(int[] array)`:
- For `add(int index, int value)`:

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

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.