

Rubber Ducky, You're The One!

5

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

```
1 public class WTF {
2     public static void main(String[] args) {
3         int i = 0;
4         while (i < 10) {
5             System.out.println("Whee!");
6             i = i++;
7         }
8         System.out.println("Done!");
9     }
10 }
```

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 **explain** what your code is supposed to do vs. what it is doing. Many times, the action of saying it out loud helps solve the problem.

Using Fields Directly vs. Using Instance Methods

6

```
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 vs. Instance Methods?

7

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

Duplicated Code: Methods

8

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 }
```

Duplicated Code: Constructors

9

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 }
```

Implementing remove

10

(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

Class CONSTANTS

11

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

Illegal Arguments

12

```
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!

Preconditions

13

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

14

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?

15

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

16

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