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
int a = 5;
Integer b = 10;
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
ArrayList<Integer> bad1 = new ArrayList<>(new ArrayList<Integer>()); // This won't compile!
ArrayList<Integer> better = new ArrayList<Integer>();
better.add(5); // We can add an 'int' to an 'Integer' ArrayList
```

Clients and Implementors

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
Clients and Implementors

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

Implementor View of ArrayList

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

What state do we keep track of? (Fields)

Classes, Objects, and Instances

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.

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

Example Class

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
```
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;
    }
}
```

Classes, Objects, and Instances

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.

```java
ArrayList<String> list1 = new ArrayList<String>();
ArrayList<String> list2 = new ArrayList<String>();
// 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.

```
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

ArrayIntList

- 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
```
1st.add(222):
```
3 | 8 | 2 | 45 | 222 | 0 | 0 | 0
```

How do we add to the end of the list?
- Put the element in the last slot
- Increment the size

Implementing add

(size = 4)
```
3 | 8 | 2 | 45 | 0 | 0 | 0 | 0
```
1st.add(222):
```
3 | 8 | 2 | 45 | 222 | 0 | 0 | 0
```

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++;
}
```
Printing an ArrayIntList

```
System.out.println automatically calls toString on the given object.
toString looks like:
1 public String toString() {
2     ...
3 }
```

Implementing add #2

```
(size = 4) 3 8 2 45 0 0 0 0
list.add(1, 222):
[size = 5] 3 222 8 2 45 0 0 0 0
```

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

```
(size = 5) 3 222 8 2 45 0 0 0 0
list.add(1, 222):
[size = 6] 3 222 8 2 45 0 0 0 0 0
```

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

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

Rubber Ducky, You’re The One!

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 }

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 |


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

1 /* Inside the ArrayIntList class... */
2 public void add(int value) {
3     this.setData(value);
4     this.size += 1;
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.setData(i, this.getData(i-1));
11     }
12     this.setData(index, value);
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

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 }

Class CONSTANTS

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 int DEFAULT_CAPACITY = 10;

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

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):
  // 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, UnsortedOperationException, 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;
}
```

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

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

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

Example ArrayList

Client View:

| 29 | 1 | 3 | 9 | 8 |

Impl. View:

| 29 | 1 | 3 | 9 | 8 |

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 |

Resize:

| 29 | 1 | 3 | 9 | 8 |

Insert:

| 29 | 1 | 3 | 9 | 8 | 2 | 1 |

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
- 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

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.