COP2800C Module 7 Practice Exercise

In this practice exercise we will continue revising our box factory application in order to practice the following new concepts from this module:

- inheritance
- interfaces

Let's start by thinking about future modifications to the application design. What if our factory suddenly needs to store other types of containers in addition to boxes? This could include bags, drums, and others. We can start the process of improving our design by supporting those other containers using **inheritance**.

Think about a box's general relationship to those other containers. A box **is a** container, with some specific attributes that differentiate it from other type of containers. A box has three primary dimensions: length, width, and height. A drum, on the other hand, might have height and diameter. A bag, at least a filled bag, can be oddly shaped depending on its contents, so it would possibly have only a volume. Regardless of those attributes, they are all containers and could be categorized using our Size and Color enumerated types.

Since we can apply a size and color enum to any of those types of containers, and because we see that there is a strong **is-a** relationship between a box and a container, it makes sense that we can generalize on the container concept by creating a Container superclass that our Box class can inherit from.

```
// class which represents a Container
public class Container {

    private Integer id = 0;
    private Size size = Size.NO_SIZE;
    private Color color = Color.NO_COLOR;

    // default constructor
    public Container() { }

    // overloaded constructor which accepts id, color, and size parameter
    public Container(Integer id, Size size, Color color) {
        this.id = id;
        this.size = size;
        this.color = color;
    }
```

I have moved all of the current instance variables from the Box into this new Container class in the file Container.java. As mentioned previously, all containers share these properties so we can create this more generalized class to store them.

Next make the necessary change to specify that the Box class inherits from the Container class:

```
// class which represents a box
public class Box extends Container {
```

Now we implement the three dimensions which distinguish a box from the other containers; I have added three new instance variables of type Double:

```
// class which represents a box
public class Box extends Container {
```

```
    // dimensions in inches
    private Double length, width, height;

    // default constructor
    public Box() {
        // call to super() is implicit here
        this.length = this.width = this.height = 0.0;
    }
```

I am making an assumption here that fractional values are possible, in the real-world we would need to verify this with the user. I note the units in a comment, if this were an international/localized application we would need to provide support for different units (we design internationalized applications in COP2805C, Advanced Java).

Default values for these instance variables are 0.0 so I have not initialized these new variables, but just to be safe I initialize them in the default constructor. Notice that the default superclass constructor does not need to be called in this constructor, it is called implicitly. If we did call it explicitly it would have to be called first, before any other code.

Now let's take a look at the overloaded constructor for the Box class:

```
    // overloaded constructor which accepts id, color, and size
    // parameters for the superclass in addition to box dimensions
    public Box(Integer id, Size size, Color color,
               Double length, Double width, Double height) {
        super(id, size, color);
        this.length = length;
        this.width = width;
        this.height = height;
    }
```

Since this is an overloaded constructor, I have to call the overloaded superclass constructor using super. And again, this has to be the first thing that happens before any other code. I pass the id, size, and color to the superclass (Container) class's overloaded constructor which will set the values of the inherited attributes. Then I set the values for the Box dimensions.

Since I moved the instance variables into the Container class, I also moved the associated accessors to that class. But since I have added three new variables to the Box class, I need to add the accessors for those variables. As before, those variables are read-only so there are no mutators, we will only set them when the Box objects are created.

```
    // dimension accessors
    public Double getLength() {
        return this.length;
    }

    public Double getWidth() {
        return this.width;
    }

    public Double getHeight() {
        return this.height;
    }
```

```
    // no dimension mutators, they are read-only
```

Take a look at the Container class's toString() method, it contains what was originally in the Box class:

```
    // toString
    @Override
    public String toString() {
        String returnStr = this.getId() + "," +
                           this.getSize().name() + "," +
                           this.getColor().name();
        return returnStr;
    }
```

Container class toString()

Now look at toString() in the Box class. This uses a neat trick for printing the state of the object. We could explicitly call the accessors for the Container class (we are inheriting them, after all) from Box's toString(). But instead, Box is letting the toString() in the Container superclass provide its own information (as a String) which is then simply prepended to Box's specific information (the dimension variables).

```
    // toString
    @Override
    public String toString() {
        String returnStr = super.toString() +
                           "(" + this.getLength() + "," +
                                 this.getWidth() + "," +
                                 this.getHeight() + ")";
        return returnStr;
    }
```

Box class toString()

We also need to change the equals method in the Box class. As with the toString, we will let the superclass tell us if the objects are equal at that level; this is essentially the same code we used in the Box class in the previous module's assignment:

```
    // check if containers are equal
    @Override
    public boolean equals(Object o) {
        boolean result = false;

        if (o instanceof Container) {
            Container c = (Container)o;

             if     (c.getId().equals(this.getId()) &&  // use .equals for wrapper
                     c.getColor() == this.getColor() && // == works for enums
                     c.getSize() == this.getSize())
                result = true;
        }

        return result;
    }
```

Container class equals()

The Container class code is called using the super reference below. Then we check our own variables at the subclass level (the three dimensions variables). Note that we cannot use == for the comparisons here since Double values are objects, we have to use the equals method provided by the Double class.

```
   // check if boxes are equal
    @Override
    public boolean equals(Object o) {
        boolean result = false;

        if (o instanceof Box) {
            Box b = (Box)o;

            if ((super.equals(b) == true) &&
                    this.getLength().equals(b.getLength()) &&
                    this.getWidth().equals(b.getWidth()) &&
                    this.getHeight().equals(b.getHeight()))
                result = true;
        }

        return result;
    }
```

Box class equals()

Now let's think about our table method. This is a useful general purpose method, we should be able to use it in other classes if we account for any changes for any differences in the data stored by those classes. To convince the other container classes to provide a table method of their own, let's create an interface which describes that method:

```
package edu.fscj.cop2800c.table;

// this interface provides a method which produces a formatted data table
// as a String
public interface TableFormatter {
    public String table();
}
```

This code resides in it's own file: TableFormatter.java, note that I have also put it in its own package: edu.fscj.cop2800c.table. This is the first time we have used more than one package for our application. I've moved all of the container/box classes down into their own sub-package named container (edu.fscj.cop2800c.container).

To use this interface, we tell the BoxFactory that it must implement it, which means that BoxFactory will have to provide an implementation of the table method (which it already does, but now we are *forcing* it to do so). Notice that we have to import the TableFormatter interface first.

```
package edu.fscj.cop2800c.container;

import java.util.Arrays;
import edu.fscj.cop2800c.table.TableFormatter;

public class BoxFactory implements TableFormatter {
```

Other notable changes for this exercise: I have added a Size enum value (Jumbo) and a new color (Yellow) to the Color enum.

I have added more data to the boxes.csv data file attached to the project and increased the MAX_BOXES constant in BoxFactory.java to handle the additional boxes.

In addition to adding more boxes and using the new Size and Color values, I have added three columns to the data file representing the three new dimension variables. Check the BoxFactory.java and BoxReader.java files to see the parsing modifications to deal with these changes.

In the main method of the BoxFactory class, test the toString method for the class and the table method, followed by a simple test of the Box/Container equals() method.