COP2800C Module 2 Practice Exercise

In this practice exercise we will design a Java program which represents a factory containing boxes of various colors. This will demonstrate the concept of **aggregation**, or "weak association" where one object contains other objects. Aggregation represents a **has-a** relationship between objects. Compare this to **composition**, or "strong association" in which the aggregated objects cannot exist without the aggregator. In this example the boxes can exist independently of the factory.

Expected output is as follows (you can use different colors if you choose to do so):

```
Box Factory contents:
First Box is a RED box.
Second Box is a GREEN box.
And the final box is a BLUE box.
```

- Start by creating a public class named BoxFactory. This means that your Java source code file must be named BoxFactory.java.

```
public class BoxFactory {
```

- Include a static integer constant which represents the total number of boxes in the factory.

```
public static final int NUMBER_OF_BOXES = 3;
```

- Declare three private Box instance variables (the Box class will be defined in this source file later in this walkthrough). Instance variables are declared at the top of the class, just after any static values and before the constructors and other methods. This is a convention we follow, not a language rule. Since they are all the same type, we can declare them in a singie statement:

```
private Box box1, box2, box3;
```

Notice that these variables have not been initialized. We *can* initialize them if we want , we would use the literal null since objects that are not instantiated take that value:

```
private Box box1 = null, box2 = null, box3 = null;
```

We will see in an upcoming module, however, that Java initializes instance variables as null for us. This *does not* apply to local variables declared inside a method, those must be initialized explicitly before using them.

- In the main method, instantiate an object of type BoxFactory. This is required since our main method is static; we cannot access our Box instance variables without an object instance which contains them. Refer to the Module 1 lecture slides for more information about static declarations.

```
public static void main(String[] args) {

    // create our Box factory
    BoxFactory boxFact = new BoxFactory();
```

- After instantiating the BoxFactory object, create three Box objects using local variables. These boxes are instantiated by calling an overloaded constructor and passing a color (we will create the Color class later in this walkthrough).

```
// instantiate local boxes
Box box1 = new Box(Color.RED);
Box box2 = new Box(Color.GREEN);
Box box3 = new Box(Color.BLUE);
```

- Now add the boxes to the BoxFactory object for the factory by assigning the local box objects to the Box Factory object using dot notation:

```
 // add the boxes to the factory
boxFact.box1 = box1;
boxFact.box2 = box2;
boxFact.box3 = box3;
```

We can assign to the private Box variables because we are in the BoxFactory class where they are declared. If we were accessing them from a different class we would have to use accessors.

- Finish your BoxFactory's main method by printing out the state of the box objects. The color is the only attribute we will use in our Box objects for now, so to display the state we simple print the box color as shown in the expected output above.

```
// Use the toString method to print box state (color)
System.out.println("First Box is a " + boxFact.box1 + " box.");
System.out.println("Second Box is a " + boxFact.box2 + " box.");
System.out.println("And the final box is a " +
                   boxFact.box3 + " box.");
```

The toString method referred to in the descriptive comment is a method we will create in the Box object; it has "magical" qualities described below which allow us to show the object state just by printing the object reference without referencing any of the attributes of that object.

- Create a Color class below your BoxFactory class. This class will encapsulate all available colors as constant integer values and also provide constant String representations. Create constants for "no color" (0), red, green, and blue (you can create more colors if desired).

Note: this class is considered a "utility" class because it has no state (no instance variables or instance methods). No objects are created from this class, we just use the public constants and methods. Do *not* declare this class as public, only one public class is allowed per source file.

```
// class to represent color values
class Color {

    // constant int representation
    public static final int NO_COLOR = 0;
    public static final int RED = 1;
    ...

    // constant String representation
    public static final String STR_NO_COLOR = "NO COLOR";
```

```
public static final String STR_RED = "RED";
...
```

- In the Color class, include a static method which converts an integer value to its String representation; the method accepts an int parameter and returns the corresponding String:

```
// convert the int representation to String
public static String colorToString(int color) {
```

- An **if** statement (or a switch statement if you're brave; we used switch statements in COP1000C) is required for this conversion. We have not covered Java if statements yet, but they are simple to implement; here is what one looks like for this method:

```
    // return value defaults to no color
String returnStr = STR_NO_COLOR;

if (color == RED)
    returnStr  = STR_RED;
else if (color == GREEN)
    returnStr = STR_GREEN;
else if (color == BLUE)
    returnStr = STR_BLUE;

return returnStr;
```

Don't forget your closing brace for this method.

- Finally, create the Box class. This will be used to create the objects that are aggregated by the BoxFactory object. Again, do not declare this class as public since only one publix class is allowed per source file.

```
// class which represents a box
class Box {
```

- Declare a private instance variable which represents the box color. We are using simple integers to represent the colors for now, in the next module we will see how to use enumerated types for a more natural representation of the colors. Here we are using the NO_COLOR constant from the Color class to initialize our instance variable.

```
// color is the only attribute
private int boxColor = Color.NO_COLOR;
```

- Declare a default constructor and an overloaded constructor. The overload accepts an int parameter for the box color.

```
// default constructor
public Box() { }

// overloaded constructor which accepts a color parameter
public Box(int color) {
    boxColor = color;
}
```

- Finally, we declare a toString method which provides the ability to create a customized String that represents the current state of an object. toString is an override of a method provided by the parent class (superclass) of the Box class. Although there is no explicit parent class that Box inherits from, it implicity inherits from the Java Object class. It turns out that every Java class inherits from the Object class. The method override replaces (or supplements) the toString method inherited from Object. If we don't override it the Object class's version prints an object reference which is not useful.

To declare an override we must enter the method header **exactly** as it occurs in the parent class.

You can see the details for the Object class by looking at the online API documentation for Java:

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Object.html

You should familiarize yourself with looking at these API pages, they are critical for learning the language in depth. Here is the toString method as declared in the Object class:

```
public String toString()
```

To properly override this method in our Box class, we must declare it exactly as shown in the Object class's declaration. If we precede the declaration with an @Override annotation, the compiler will verify that the method has been properly overridden:

```
@Override
public String toString() {
```

What can we do in this method? We want to create a string which represents the current state, which means displaying the current value of the instance variable. Here we are calling the static colorToString method provided by the color state. We did not call this method toString because we did not want confuse it with an actual toString method; remember that the Color class has no state and we do not create objects from it. The overridden toString method in our Box class is a true instance method; an object is required in order to use it.

```
        return Color.colorToString(boxColor);
    }
```

Note that we never print anything from toString, we return a String and let its caller do the printing. Remember our println statements in the BoxFactory's main method?

```
System.out.println("First Box is a " + boxFact.box1 + " box.");
```

Here is the beauty of the toString method. We are not calling toString explicitly in this operation. Java looks for a toString method in our class and, if it finds one, calls it for us! We could call it if we wanted to:

```
System.out.println("First Box is a " +
                   boxFact.box1.toString() + " box.");
```

But why waste the keystrokes when the language does it for us?

This is the end of the Box class, close it off with a closing brace }

Once you have completed this practice exercise on your own, take a look at the posted solution.

If you're curious, here's what the high level UML class diagram looks like for our application. The relationship between the BoxFactory and Boxclass is **aggregation** ("has-a"); the Box objects created in our main method can exist independently of the BoxFactory object we created, but they are aggregated in that class.

The Color class is used by the Box class, so there's an association but it is weak since we do not create any Color objects; in this case I've used a dashed line to indicate there's a simple **dependency**, a weaker form of an association**,** because Box calls the static method in Color.