COP2800C Module 6 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:

- wrapper classes
- selection structures
- repetition structures
- exceptions

We will also work with a comma-separated values (CSV) file to read our data from an external source instead of creating it directly in our application.

We will continue working with JAR files and, as with Module 5, we will bundle our source and class files into a JAR file that can be executed from the command line (and submitted as your assignment solution).

- Start by making the following changes to the Box class:

  Add an Integer id field. If we were storing this data to a relational database, this would be our unique ID which could be used as a primary key for a "Box" table. Use Integer (not int) as the data type, this is our wrapper class.

  Add a boxSize field. Create an enum for this which supports three sizes: SMALL, MEDIUM, LARGE, and also provides a "NO_SIZE" value.

```
public class Box {

    private Integer id = 0;
    private BoxSize boxSize = BoxSize.NO_SIZE;
    private Color boxColor = Color.NO_COLOR;
```

  Add an accessor for the id field, but not a mutator – this will be a read-only field that is set when the object is instantiated.

  Add an accessor for the boxSize field, but no mutator, it is also read-only.

  Modify the overloaded constructor for the Box to accept three parameters instead of one to initialize the new fields.

```
// overloaded constructor which accepts id, color, and size parameter
public Box(Integer id, BoxSize boxSize, Color boxColor) {
    this.id = id;
    this.boxSize = boxSize;
    this.boxColor = boxColor;
}
```

Modify the equals and toString methods to include the new fields as well. The equals rules may be too complicated for a ternary expression, here you see it has been converted to an if-else. Notice how I have continued the line and aligned the expressions in the if statement for readability:

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

            if      (b.getId() == this.getId() &&
                     b.getBoxColor() == this.getBoxColor() &&
                     b.getBoxSize() == this.getBoxSize())
                result = true;
        }
```

- Now modify the BoxFactory class.

  First, we will delete the NUM_BOXES constant. We are going to read our data from a file, so we do not know in advance how many boxes will be read, but we need an upper bound so we can allocate memory for our array. I've created a constant to indicate the factory can store no more than 10 boxes (it's a small factory).

```
    public class BoxFactory {

        public static final int MAX_BOXES = 10;
```

  When we instantiate the factory, we will tell it how many boxes were read in. Allocate the array's memory by using that information. Change the overloaded BoxFactory constructor as follows:

```
  // overloaded constructor - create array with boxCount Box elements
  public BoxFactory(int boxCount, Box[] boxes) {
      this.boxes = new Box[boxCount];
      for (int count = 0; count < boxCount; count++) {
          this.boxes[count] = boxes[count];
      }
  }
```

  In the BoxFactory main method, we are going to make some big changes. First, delete any code that is created individual boxes, testing the accessors or mutators, and testing the equals method. Then replace it with the following code:

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

      String[][] inputArray = BoxReader.read(MAX_BOXES);
```

  We start the program by reading the data in. I have provided a BoxReader.java file that you can incorporate into your application; we are calling the "read" method in that class and providing it with the maximum number of boxes to read from the file. Here's what that file looks like:

```
    ID,SIZE,COLOR
    1,small,red
    2,small,blue
    3,small,green
    4,medium,green
```

```
5,medium,red
6,large,red
```

This is a CSV, or comma-separated value file. Each piece of data belonging to a single object is separated by a comma. There is a header at the top of the file, also comma-separated, that we need to be careful to skip so we don't include it as valid data.

In data science, this data is considered to be in a tabular format where each row is an observation and each column is a variable. There are many ways to represent a table in a file, CSV is one very common representation. Most languages, including Java, provide libraries to work with these files. Since the Java version of these libraries use more advanced data structures that we have not covered yet, the BoxReader class uses very basic file input operations to parse the data and populate a simple 2D array.

After the read method has been called, our data is loaded in the array variable inputArray. Now we instantiate a local array named "boxArray" that will contain the number of boxes that were read from the CSV file. Note the condition in the for loop below; we look for an uninitialized element (inputArray[rows][0] – the first column in the current row) to indicate we are done populating this local array.

Notice the use of the Integer wrapper class's "parseInt" method, this is how we can convert a String to an Integer. I have also created two methods in the enumerated types which behave in a similar fashion; BoxSize.parseSize converts the String data read in from the CSV file into a BoxSize enum, and Color.parseColor does the same for the Color enum. We will look at these methods in more detail below.

```
// create an empty array with the maximum specified elements
Box[] boxArray = new Box[MAX_BOXES];
int boxCount = 0;  // may be partially filled

// fill the array with up to the number of elements that were read
for (    int rows = 1;  // skip the header!
         rows < MAX_BOXES && inputArray[rows][0] != null;
         rows++) {

    Integer newBoxId = Integer.parseInt(inputArray[rows][0]);
    BoxSize newBoxSize = BoxSize.parseSize(inputArray[rows][1]);
    Color newBoxColor = Color.parseColor(inputArray[rows][2]);

    boxArray[boxCount++] = new Box(newBoxId, newBoxSize, newBoxColor);
}
```

We now have the data necessary to instantiate our factory using our modified overloaded constructor. We pass the number of data elements and the local array:

```
// create the Box Factory's array with only the populated elements
BoxFactory boxFact = new BoxFactory(boxCount, boxArray);
```

Now let's print our data. We haven't modified the toString, we want to continue allowing it to do it's magic and call the Box class's toString. Instead a table method has been added which will "pretty-print" our data in a tabular form:

```
// toString will display unformatted data, don't use it here
```

```
// Instead, call the table method to format in a nice tabular form
System.out.println(boxFact.table());
```

Here is the table method. We are using the String class's format method (similar to the System.out.printf method discussed in Module 3, and briefly mentioned in Module 3) to format our data in a fixed-width format. The %-12s format specifiers tell the method for format the strings with a 12-character, left justified font. Notice how I have continued the line and aligned the arguments to the format method for readability; otherwise this line would be much too long.

```
// return contents in tabular format
public String table() {
    final String DATAFORMAT = "%-12s%-12s%-12s\n";
    // header
    String retString = "Box Factory Contents:\n"; // start here
    // data headers
    retString += String.format(DATAFORMAT, "ID", "SIZE", "COLOR");
    // data
    for (int boxCount = 0; boxCount < this.boxes.length; boxCount++) {
        String boxStr = String.format(DATAFORMAT,
            this.boxes[boxCount].getId(),
            this.boxes[boxCount].getBoxSize().name(),
            this.boxes[boxCount].getBoxColor().name());
        retString += boxStr;
    }

    return retString;
}
```

Here's what the formatted output looks like (using a fixed width Word font):

```
Box Factory Contents:
ID          SIZE        COLOR
1           SMALL       RED
2           SMALL       BLUE
3           SMALL       GREEN
4           MEDIUM      GREEN
5           MEDIUM      RED
6           LARGE       RED
```

Here's the parseColor method that has been added to the Color enum:

```
public static Color parseColor(String colorStr) {
    Color retVal = NO_COLOR;
    switch (colorStr.toLowerCase()) { // convert to all lower case for comparison
        case "red":
            retVal = RED;
            break;
        case "blue":
            retVal = BLUE;
            break;
        case "green":
            retVal = GREEN;
```

```
                break;
        }
        return retVal;
    }
```

We could use an if-then structure here, of course, but because Java is one of the few languages which allow a switch statement to use Strings, so let's take advantage of it.

The new BoxSize enum has a similar method:

```
    public static BoxSize parseSize(String sizeStr) {
        BoxSize retVal = NO_SIZE;
        switch (sizeStr.toLowerCase()) { // convert to all lower case for comparison
            case "small":
                retVal = SMALL;
                break;
            case "medium":
                retVal = MEDIUM;
                break;
            case "large":
                retVal = LARGE;
                break;
        }
        return retVal;
    }
```

- Finally, let's take a look at the BoxReader class.

  We have a constant which describes the number of columns (variables). We need this to declare our 2D array.

  There are a lot of imports in this file, we need to pull in the Scanner and File objects to declare the necessary objects to read from the file. When we are creating a file object we have to catch the IOException since bad things can happen when dealing with files (doesn't exist, no permissions, etc.) so we have to import that as well.

  We could just import java.io.*; for the File and IOException classes but that's not a best practice and is inefficient. Instead, we want to be specific about what we import and only bring in the classes we need (especially since there are only two coming from that package).

```
package edu.fscj.cop2800c;

import java.io.File;
import java.io.IOException;
import java.util.Scanner;

public class BoxReader {

    public static final int BOX_DIMENSIONS = 3;

    // read data from a CSV file, return a 2D array
    public static String[][] read(final int MAX_BOXES) {
```

```
        final String FILENAME = "boxes.csv";
        String[][] boxes = null;

        // create a file instance
        File file = new File(FILENAME);
        try (Scanner input = new Scanner(file);) {
             boxes = new String[MAX_BOXES][BOX_DIMENSIONS];
             for (int rows = 0; rows < MAX_BOXES; rows++)
                 for (int cols = 0; cols < BOX_DIMENSIONS; cols++)
                     boxes[rows][cols] = null;

              // read data from file
              int rowCount = 0;
              // constants for the columns
              final int BOX_ID_COL = 0;
              final int BOX_SIZE_COL = 1;
              final int BOX_COLOR_COL = 2;

              // loop while there is more input
              while (input.hasNext()) {
                    String line = input.nextLine();
                    String[] sp = line.split(",",BOX_DIMENSIONS);
                    boxes[rowCount][BOX_ID_COL] = sp[BOX_ID_COL];
                    boxes[rowCount][BOX_SIZE_COL] = sp[BOX_SIZE_COL];
                    boxes[rowCount][BOX_COLOR_COL] = sp[BOX_COLOR_COL];
                    // increment rowCount to get ready for the next row
                    rowCount++;
              }
        } catch (IOException ex) {
              System.err.println("Exception! " + ex);
        }

        return(boxes);
    }
}
```