Lab 12: Polymorphism in Action

Objective

This lab will utilize our **Shape** example again. We will demonstrate polymorphic behavior by implementing a similar method several different ways and showing how polymorphism allows us to work with many different objects in a common and flexible manner.

Overview

In this lab you will:

- Add new behavior to the **Shape** hierarchy
- Create an array of many shape objects
- Test your new functionality

Step by Step Instructions

Exercise 1: Creating polymorphic methods

1. For this lab, you will be working in the **com.lq.exercises** package of the **ClassExercises** project.

2. When you created the **Shape** class in the previous lab, it did not explicitly inherit from any other class. In this case, it implicitly inherits from **Object**. We will experiment with one of the methods that all of our shapes inherit from **Object** which is called `toString()`. This is a method that provides a default way of printing out information about an object as a **String**. The default behavior of the method provides some basic information.

3. To test this behavior, create a new class in the **com.lq.exercises** package named **ExercisePrint**. Ensure that this method has a `main()` method.

4. Create an array of **Shape** objects and place each of your shape types in the array. [Remember, the **Shape** class is abstract and cannot be created as a stand alone object.]
    a. Add constructors to Rectangle, Square, Cube, Box and Circle which accept name and color as well as the specific attributes for that Shape.

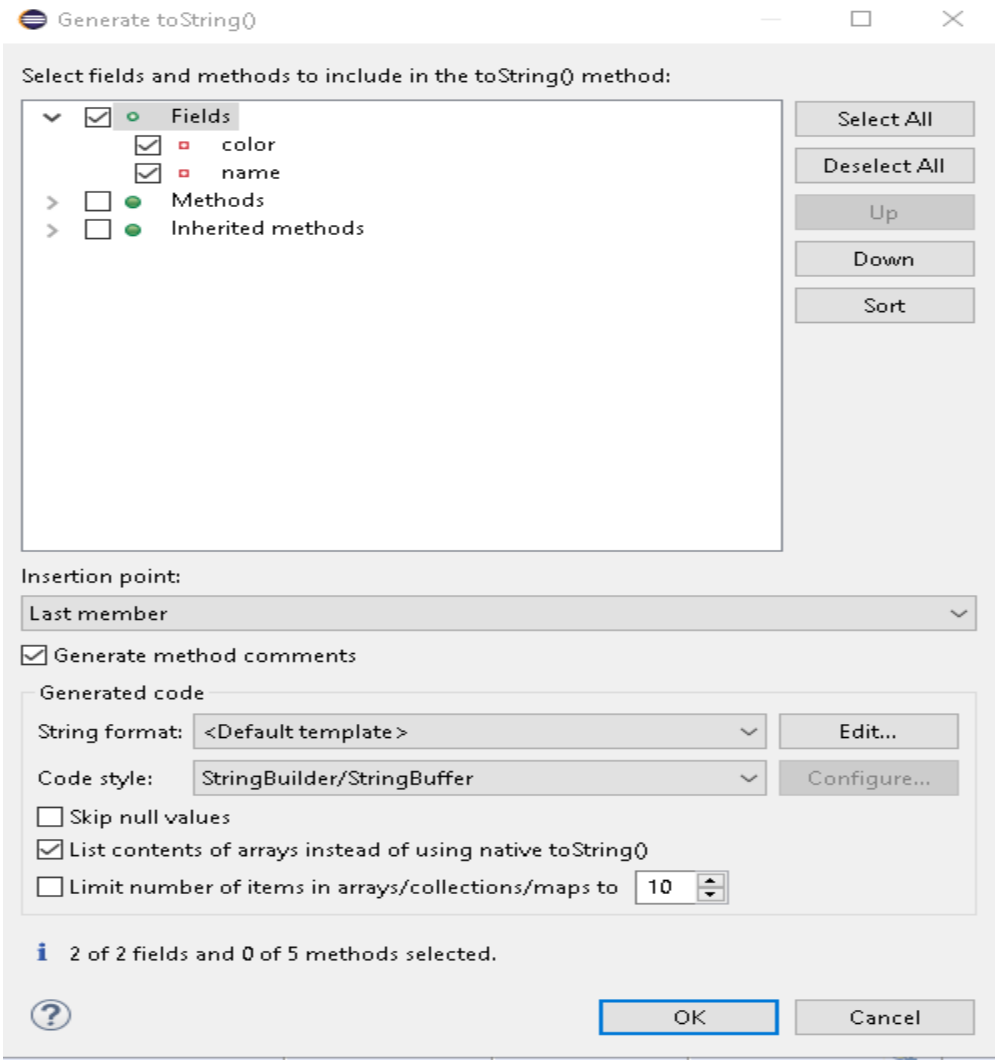
5. Loop through your array of objects and call the `toString()` method for each object in the array.

```
for(Shape shape: shapes) {
    out.println( shape );
}
```

6. Run the program and observe the output. Notice that the inherited (from Object) `toString()` method just provides some system information about the type and location of the object in memory. Not very useful! An example below.

```
com.lq.exercises.Box@27562756
com.lq.exercises.Cube@29322932
com.lq.exercises.Circle@2f6a2f6a
com.lq.exercises.Rectangle@31393139
com.lq.exercises.Square@32b732b7
```

7. In your **Shape** class, provide a toString() method using the Source -> Generate toString wizard. Make sure the Fields box is checked, the insertion point is set to “Last member”, the Generate method comments box is checked, The String format is “<Default template>”, and the Code Style is set to “StringBuilder/StringBuffer”. It should look like this:



8. If you have created toString() methods for your Shapes in earlier labs, you may have to recreate them now that we are extending Shape, with color and name;

9. Re-run your test program. Now the output is coming from the `toString()` method defined in **Shape**. We are already seeing the benefits of polymorphism. In the previous run of the method, the only behavior we had available was the default system behavior. Now we have customized that functionality and will utilize this new method for any object of type **Shape**.

10. Let's extend this concept even further. In the **Box** class, add an implementation for `toString()` using the Source->Generate toString() wizard. Add in the Inherited methods for getColor() and getName(); Your code should look like this:

```java
/* (non-Javadoc)
 * @see java.lang.Object#toString()
 */
@Override
public String toString() {
    StringBuilder builder = new StringBuilder();
    builder.append("Box [length=");
    builder.append(length);
    builder.append(", width=");
    builder.append(width);
    builder.append(", height=");
    builder.append(height);
    builder.append(", getColor()=");
    builder.append(getColor());
    builder.append(", getName()=");
    builder.append(getName());
    builder.append("]");
    return builder.toString();
}
```

11. Re-run your test program. Notice how the output changed. When we encounter a **Box** or **Cube**, `toString()` from the **Box** class is used. Otherwise, `toString()` from **Shape** is used. This is polymorphic behavior. Even though we are working with an array of **Shape** objects, the behavior from the class that defines the actual object type in the array is used when methods are called.

12. Extend this concept by writing `toString()` methods for **Circle** and **Rectangle**. [Hint: When you created **Circle**, you may not have created a get and set method for `radius`. Do that now in order to implement `toString()` appropriately.]

    Your output should look similar to the followingwhen done:

```
Rectangle [length=5.0, width=4.0, color=Blue, name=Rectangle1]
Rectangle [length=8.0, width=8.0, color=Red, name=Square1]
Box [length=5.0, width=5.0, height=5.0, color=Yellow, name=Cube1]
Box [length=12.0, width=15.0, height=6.0, color=Orange, name=Box1]
Circle [radius=3.0, color=Lavender, name=Circle1]
```

    This is polymorphism at work! We have a container that holds many different object types but as we call methods on those objects, the correct implementation of that method is called for the appropriate object.