CS102 **Lab03 ~ Get into shape** Fall 2019

Introduction

This assignment asks you to design, implement and explore simple class hierarchies, abstract classes and interfaces.

Note: this assignment has nothing to do with drawing graphics in Java... everything is done on the console as usual.

Create two projects, one for part (a) and another for part (b).

(a) Shape up

(1) Design a class hierarchy to include classes *Shape*, *Rectangle* (with *int* sides, *width* and *height*), *Circle* (with *int radius*) and *Square* (with *int side*). Shape should be abstract with method *double getArea()* and Square should inherit *getArea()* from Rectangle.

(2) Create another class, *ShapeContainer*, to hold a set of shapes. It should have methods *void add(Shape s)* and *double getArea()* and *String toString()*. Write a *ShapeTester* class with a menu that allows the user to create an empty set of shapes (ShapeContainer), add as many circle and rectangle shapes to it as they wish, compute & print out the total surface area of the entire set of shapes, and print out information about all of the shapes in the container by calling the toString() method for each shape. Experiment. Try to predict what would happen when you (i) comment out the getArea() method of the Circle class, and (ii) also make the Circle class abstract, before finally (iii) creating an instance of the (now abstract) Circle class to add to the shapes collection. Test your predictions.

(3) The customer is impressed with your work so far, and so asks you to extend the program. They want shapes to be locatable (i.e. to have an *x, y* location, and *getX()*, *getY()* and *setLocation(x, y)* methods). As a good designer you decide to first create a *Locatable* interface with these methods, then have the *Shape* class implement it. In this way all shapes automatically become locatable.

(4) Impressed, the customer wants even more! This time they ask for shapes to be *Selectable*, so you again start by creating a Java interface, having *boolean getSelected()* and *setSelected(boolean)* and *Shape contains(int x, int y)*. Unfortunately, this time you are <u>*not allowed*</u> to change the *Shape* class. Modify your other classes so that each shape added to the *ShapeContainer* is *Selectable*. Change the *toString()* methods of each shape class so they show whether the shape is selected or not. Add another option to your ShapeTester menu that allows the user to find the first Shape that contains a given x, y point and, afterwards, toggle its selected state. Provide another menu option that removes all selected shapes from the set of shapes. Good design practice suggests you should ask the *ShapeContainer* object to do the work of finding the first Shape containing the given point and of removing all selected shapes (rather than trying to do the work yourself in the

ShapeTester class, which might/would require knowledge of the insides of the *ShapeContainer* class).

(b) Iterating IntBags

The Java API includes an interface called *Iterator* that is designed to retrieve each of the objects from a set of objects, once and once only. This allows programs to process each item in the set without being aware of the underlying implementation (which means it can be changed at will without affecting the client program.) Your task is to provide this interface for your *IntBag* class (from Lab01). *Note*: The *IntBag* class was designed to store primitive *int* types, rather than Java *Object* types. Since the *Iterator next()* method must return an *Object*, you will need to return the corresponding *int* in an *Integer* wrapper. Clearly, this is not an entirely satisfactory solution and you might consider extending *Iterator* to produce *IntIterator*, which provides an additional *int nextInt()* method.

There are several ways you might think of coding this problem. Clearly you must have a value (say *index*) that keeps track of position of the next item to be retrieved from the *IntBag* collection. This must be initialized to zero when an iteration begins, and be incremented each time a call is made to get the next element (such call returning the element at the *index* location within the set.) The *hasNext()* method simply determines whether the *index* value is less than the number of elements in the set or not. Putting this code inside the *IntBag* class is problematic; how can index be reset to zero so that the elements can be iterated through several times, and how can multiple independent iterations through the set be managed?

A simple solution to this is to write a new class, say, *IntBagIterator*, which implements Iterator. This class need have only two properties; *aBag* (a reference to the *IntBag* it is to iterate through) and *index* (the position of the next element to return from *aBag*). Code this class and test it using a class, *TestIterators*, which includes the following code:

```
Iterator i, j;

i = new IntBagIterator( bag );

while ( i.hasNext() )
{
        System.out.println( i.next() );

        j = new IntBagIterator( bag );
        // j = bag.iterator();

        while ( j.hasNext() )
        {
                System.out.println( "--" + j.next() );
        }
}
```

Notice that i & j are of the interface type, *Iterator*. This is valid since *IntBagIterator* implements *Iterator* and so *is_a Iterator*.

Finally, add a method *Iterator iterator()* to your *IntBag* class. This should create an instance of *IntBagIterator* and return it as the interface type. Test this method by uncommenting the commented line in the above code and then commenting out the line above it. *Note:* can you make your *IntBag* implement Java's *Iterable* interface now?

Java's *ArrayList* class has a similar method that allows you to iterate through its elements. Interestingly, there is no (immediately obvious) class in the Java API that corresponds to our *IntBagIterator* class. It actually has such a class, but nested inside *ArrayList*. Try making your *IntBagIterator* an inner class of your *IntBag* too.