**CS 273 Laboratory 8: Methods and Classes**

This lab gives you practice creating a simple Java class.

**Preliminaries**

In this laboratory, you will create a Java class, Die, that models a six-sided die. Each part of this lab deals with two classes: a Run*X* class, which is provided, and a Die class, which you will create. In each successive part, the Run*X* class makes calls to additional Die methods. You will need to implement these methods as the lab progresses.

**Do not modify any of the Run*XX*.java files for this lab. You should only modify Die.java** even when BlueJ tells you that there is an error in one of the Run*XX*.java files.

There are several RunXX classes. Each RunXX corresponds to the class you should use to run the main method for a given checkpoint. For instance, Run1 should be used for checkpoint 1, Run2 should be used for checkpoint 2, and so on.

**Laboratory**

You will be implementing the Die class in this lab. The following chart provides an overview of the attributes (instance variables) and actions (methods) of the Die class. This is for reference only. You do not need to remember everything in this chart.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Die** | | | | | | |
| **Attributes (Instance Variables)** | **Associated Checkpoint** | **Public?** | **Name** | **Type** |  | |
| Checkpoint 1 | no | currentValue | int |  | |
| Checkpoint 3 | no | size | int |  | |
| Checkpoint 4 | no | xCoord | int |  | |
| Checkpoint 4 | no | yCoord | int |  | |
|  | | | | | | |
| **Actions (Methods)** | **Associated Checkpoint** | **Public?** | **Name** | **Return Type** | **Input Parameters** | |
| **name** | **type** |
| Checkpoint 1 | no | reRoll | void |  |  |
| Checkpoint 1 | yes | paint | void | g | Graphics |
| Checkpoint 1 | yes | roll | void | g | Graphics |
| Checkpoint 1 | yes | Die (constructor) | N/A | (none) |  |
| Checkpoint 2 | no | drawSpot | void | g  xPos  yPos | Graphics  int  int |
| Checkpoint 3 | yes | setSize | void | newSize | int |
| Checkpoint 3 | yes | getSize | int |  |  |
| Checkpoint 4 | yes | Die (constructor) | N/A | xCoord  yCoord | int  int |
| Checkpoint 5 | yes | getCurrentValue | int |  |  |
| Checkpoint 5 | yes | toString | String |  |  |
| Checkpoint 5 | yes | equals | boolean | other | Die |

**Background Information:** Notice the five other classes in your BlueJ Project: Run1, Run2, Run3, Run4, and RunAbstract. Each of these classes, already implemented by Dr. Vegdahl, does the following:

* Creates a Die object by calling your Die constructor.
* Whenever the user presses the Roll button, it calls the Die object's roll method (which you will create below).
* Whenever it needs to repaint itself, it calls the Die object's paint method (which you will create below), and displays a text message that says how many times the die has been rolled.

At this point, if you were to compile your entire BlueJ project, you would have a number of compilation errors in the various Run classes. These errors are because you haven’t finished writing enough code yet to make your Die class compatible with them. Do not change the Run classes; keep coding.

**Part 1: Create a simple Die class**

* 1. Open the project.
  2. Modify the file Die.java (which is presently an empty class definition) so that the class Die contains the following internal state, as a private instance variable.
* currentValue that tells which face of the die is showing (This should always be in the range 1 through 6.)

1. Create a constructor that does not take any parameters. The constructor should only do one thing: call a method called reRoll(). You will implement the reRoll() method in the next step.
2. Create a private method, reRoll(), which updates the currentValue of the die with a new random value between 1 and 6, inclusive. You may find the Math.random() method useful for generating random numbers.

**Hint**: If you multiply a random number generated by Math.random() by 6, this new number will be in the range [0, 6.0), meaning it includes 0.0 and excludes 6.0.

1. Create a public method, roll(). It should not return anything, and it takes a single Graphics object as a parameter (you will need to import java.awt.Graphics first). The roll() method must first call the reRoll() method, and then it should call the paint() method so that the die is drawn to the screen. You will write the paint() method in the next step.

**Design Note**: There is a separate roll() and reRoll() method because one may want to call reRoll() in several places inside the Die class (reroll() is a private method), but outsiders may only roll() the die so that the change is reflected on the screen (roll() is a public method).

1. Create a public method, paint(), which draws the die onto a Graphics object. The method should not return anything. It takes a single parameter: a Graphics object. The paint method should have the following behavior:

* (You will need to import java.awt.Color first.)
* It must draw a single *white* 50 x 50-pixel square with a *black* border. This is the die. If the input parameter is g, then the method may call the drawing methods on g by using the syntax g.drawRect(....arguments here....);.
* The die should be drawn so that the upper-left corner of the drawn die is at (0,0) – i.e., the upper-left corner – of the canvas.
* The currentValue should be printed as *black* text somewhere inside the die. To do so, currentValue must be converted to a String. Integer.toString(int) may help you.
* To perform error-checking, the paint method must check that the Graphics object is not null before attempting any drawing on the Graphics object.

1. Compile the Run1 class, right-click on Run1, and call its main method.

**checkpoint 1 (20 points): Show your lab instructor or assistant the executing program.**

**Part 2: Modify the paint method in the Die class so that it draws a die with spots (a.k.a., pips)**

1. Create a private method, drawSpot(), that takes three parameters: an x-coordinate, a y-coordinate, and a Graphics object. It should draw **ONE** filled circle with a diameter of 10 pixels, **centered** at the given x-coordinate and y-coordinate.

For example, if you pass this method an x-coordinate of 0 and a y-coordinate of 0, then a pip should be drawn with its center at the top left corner of the die.

1. Modify the paint method so that it draws a die with filled black spots **rather than a single digit**. For example, if currentValue is 2, your paint method should display 2 dots. You will likely have to treat each of the 6 possible die-values as a special case. A switch statement may be useful here. The paint() method should call (invoke) the drawSpot() method to draw each spot on the die.

**Note:**

* **The pips must not touch the edges of the die** and must be drawn in traditional dice format:      
* Again: each call to drawSpot() should only paint one spot. Your code should call drawSpot() multiple times when you want to draw multiple spots.

**checkpoint 2 (20 points): Call the main method of the Run2 class to show your lab instructor or assistant the executing program.**

**Part 3: Add a method that allows you to change the size of a die**

**Background Information:** The Run3 class implements the same behavior as before. In addition, it changes the size of the die. It does this by invoking the setSize() method in this line of code:

myDie.setSize(150);

The + Size button on the interface increases the size of the die and the - Size button decreases the size of the die. Each click of the button changes the die size by 10 pixels in each direction.

As you increase or decrease the size of the die by pressing on the buttons, **the die and its pips should scale in size accordingly.**

1. Because Die objects may now have different sizes, you need to store the size of a die as part of its state. Thus, you should add a private instance variable to your Die class that gives its size in pixels.
2. Add a public method, setSize(), that takes one int parameter, and sets the size of the Die to the value specified by the parameter. It should not return a value.
3. Add a public method, getSize(), that takes no parameters and returns the size variable you created in step 1 above.
4. The initial size of the die should be 50 before setSize() is ever called. This means that the Die constructor should call setSize(50).
5. Modify the paint() method so that it draws the die as an NxN-pixel square where N is the value of the new instance variable. Next, modify drawSpot() so that the spots on the Die scale along with the die – i.e., when the die gets bigger, they should get bigger and their positions should “stretch out”. E.g., if there is a spot in the center of the die (like when you roll a 1, 3, or 5), it should stay in the center regardless of the die’s size.

**Requirement:** the scaling of both position and size of the pips should happen within drawSpot() – the calls to drawSpot() inside paint() should remain exactly as they are!

**Hint:** since you originally coded the positions of the pips for a 50x50 die, the scale factor (i.e., position multiplier) for the pips will be the die’s current size divided by 50.

**Hint 2:** You must have the division part of the scale at the very end of the math or you will lose values In the Int conversion for drawSpot().

**checkpoint 3 (20 points): Show your lab instructor or assistant the executing program.**

**Part 4: Draw a second die at a different position**

The instructor’s code in Run4 will attempt to create \*two\* dice this time. The first die will be at the same position as before -- i.e., top-left corner at (0,0) – whereas the second die’s top-left corner will be given an x-coordinate of 210 pixels and a y-coordinate of 50 pixels. Here are the lines of code in Run4 that make the two Die objects:

myDie = new Die(0, 0);

myDie2 = new Die(210, 50);

Your job is to modify your Die class to make those things happen.

* 1. Give the Die two more private instance variables:
* an x-coordinate to represent the upper left-hand x-coordinate of a die
* a y-coordinate to represent the upper left-hand y-coordinate of a die

2. Create a second constructor that takes two parameters: the x and y positions. The constructor has two steps.

* First, it should call your other constructor (so you don’t have to copy-paste all the code from it!) with this code: this();
* Second, it should initialize the two new instance variables using the two parameter values.

3. Change your code in paint() so that the second die shows up at the correct position.

**Requirement:** adjusting the position of the pips should happen within drawSpot() – the calls to drawSpot() inside paint() should remain exactly as they are!

**Hint:** paint() anddrawSpot() can directly access the x- and y-coordinate instance variables you just made; you don’t need to pass them in as parameters.

**checkpoint 4 (20 points): Show your lab instructor or assistant the executing program.**

**Part 5: Get the die value, convert the die into a character-string, and identify whether two dice are equal**

**Background Information:** The Run5 class prints a message stating whether the current dice denote "doubles", that is, if the two die have the same value. You might need to make the window bigger to see where it prints this message – it’s at the very bottom of the interface.

Due to this new functionality, it is necessary to implement three additional methods in the Die class.

* 1. Create a public method, getCurrentValue(), that takes no parameters and returns the value of currentValue of a Die object.
  2. Create a public method, toString(), that takes no parameters and returns a String object. The String object returned should be a String containing a single character, where the character corresponds to the currentValue of the Die object. For example, if currentValue is 4, then the String returned should be "4". Here is a method for converting an int to a String: Integer.toString(int)
  3. Write a public method, equals(), that takes a Die object as its only parameter. It should return a boolean value that tells whether the face value, or currentValue, of “this” Die and the second Die (passed in as a parameter) are equal. The Run5 class determines if two dice are equal by invoking the equals operation, as in:

if (myDie.equals(myDie2)) ...

**Java fact:** actually, every object type in Java (including built-in object types like Strings and user-defined object types like the Dies in this lab) automatically has an equals() method. By default, it is defined like this:

public boolean equals(Object obj) {

return (this == obj);

}

In other words, it just uses == to compare the two objects. Java’s default behavior for the == comparator basically asks, "are they the same object?"; in other words, are they at the same location in memory. Because we are comparing two separate Die objects, this default equals() method (or just doing myDie == myDie2) will always return false. But if you define an equals() method in the Die class, it will overwrite the default equals() method for all Die objects, and you can make it do whatever you want!

Your task is to write an equals() method that returns true if the dice are showing the same value.

Use getCurrentValue() to get the current value of the other die. It’s not actually necessary in this case because private instance variables are accessible to objects of the same object type, but it’s a reminder that you will usually need to use getter methods to access the private instance variables of other objects.

**checkpoint 5 (20 points): Show your lab instructor or assistant the executing program.**