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

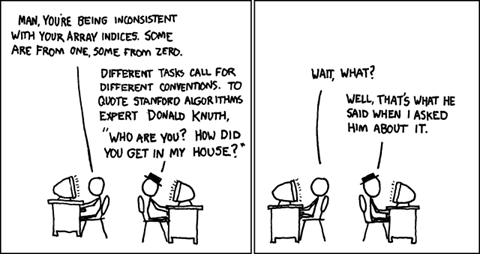
This lab gives you practice creating a simple Java class.

**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.

****

**www.xkcd.com**

**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** | **Associated Checkpoint** | **Public?** | **Name** | **Type** |  | |
| Checkpoint 1 | no | xCoord | int |  | |
| no | yCoord | int |  | |
| no | currentValue | int |  | |
| Checkpoint 4 | no | size | int |  | |
|  | | | | | | |
| **Actions** | **Associated Checkpoint** | **Public?** | **Name** | **Return Type** | **Input Parameters** | |
| **name** | **type** |
| Checkpoint 1 | no | reRoll | void |  |  |
| yes | paint | void | g | Graphics |
| yes | roll | void | g | Graphics |
| yes | Die (constructor) | N/A | xPos  yPos | int  int |
| Checkpoint 2 | no | drawSpot | void | g  xPos  yPos | Graphics  int  int |
| Checkpoint 3 | yes | getCurrentValue | int |  |  |
| yes | toString | String |  |  |
| yes | equals | boolean | other | Die |
| Checkpoint 4 | yes | setSize | void | newSize | int |
| yes | getSize | int |  |  |

**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 private instance variables.
* x-coordinate to represent the upper left-hand x-coordinate of a die
* y-coordinate to represent the upper left-hand y-coordinate of a die
* currentValue that tells which face of the die is showing (This should always be in the range 1 through 6.)

1. Create a public constructor that takes two parameters: the x and y positions. The constructor has two steps. First, it should initialize the instance variables using the two parameter values. Second, the constructor should call the reRoll method to initialize the currentValue instance variable. You will implement the reRoll method in the next step.
2. Create a private method, reRoll, which simply updates the current value 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.

**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.
* Whenever it needs to repaint itself, it calls the Die object's paint method, 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.

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:

* 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 x and y-coordinate instance variables dictate the upper-left corner of the drawn die.
* 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. Create a public method, roll. It should not return anything, and it takes a single Graphics object as a parameter. The roll method must call the reRoll method to set currentValue to a number between 1 and 6, inclusive. After calling the reRoll method, it should call the paint method so that the die is drawn to the screen.

**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. It may be necessary to add one or more import statements to the top of your Die.java file before it will compile.
2. Compile the Run1 class, right-click on Run1, and call its main method.

**checkpoint 1 (25 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: a 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.

**IMPORTANT:** It is essential that you implement this so that drawSpot expects the x and y coordinate passed to the method to be specified *with respect to the top-left corner of the die* rather than with respect to the upper left corner of the canvas. For example, if x and y are passed to this method as (0,0), then the pip should be drawn in the top left corner of the die, not of the canvas. If this instruction is confusing to you, ask now. Incorrect code will have to be rewritten.

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:      .
* If you draw more than one spot in the drawSpot method, you will not get credit for this checkpoint. Paint should call drawSpot multiple times to draw multiple spots.
* If every call you make to drawSpot has arithmetic in the arguments, you will not get credit for this checkpoint. The arithmetic should be done in the drawSpot method.

**checkpoint 2 (25 points): Call the main method of the Run2 class to show your lab instructor or assistant the executing program.**

**Part 3: Get the die value, convert the die into a character-string, and identify whether two dice are equal**

**Background Information:** The Run3 class does more than in the previous steps. In addition to the previous behavior:

* it creates a second Die object (in a different location than the first) and rolls both when the Roll button is pressed
* it prints the minimum two-die sum it has seen so far
* it prints the maximum two-die sum it has seen so far
* it prints a message stating whether the current dice denote "doubles", that is, if the two die have the same value.

Due to its increased 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 a Die object. For example, if currentValue is 4, then the String returned should be "4".
  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 currentValues, of “this” Die and the parameter Die are equal. The Run3 class determines if two dice are equal by invoking the equals operation, as in:

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

Java’s default behavior for the equals method is "are they the same object?"; in other words, are they referenced by the same location in memory. Because two separate Die objects are used, the default equals method will always return false. It is your job to create an equals method that returns true if the dice are showing the same value.

**checkpoint 3 (25 points): Show your lab instructor or assistant the executing program.**

**Part 4: Add a method that allows you to change the size of a die**

**Background Information:** The Run4 class implements the same behavior as before. In addition, it changes the size of the second die. It does this by invoking the setSize method, as in:

myDie2.setSize(100);

+ Size button on the interface increases the size of the second die and the - Size button decreases the size of the second die. The size of the first die does not change. Each click of the button changes the die size by 10 pixels in each direction.

The Run4 class will set the initial size of the second die to 150. 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. 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. The sizes and positions of the spots on the Die should be adjusted accordingly.
5. The initial size of the die should be 50 before setSize is ever called. This means that the Die constructor should include the following line:

this.setSize(50);

In completing the work above, you will need to account for the radius of a spot. Do not pass this value into every call of drawSpot. Instead, do the appropriate arithmetic within the method (like you did for checkpoint 2).

**checkpoint 4 (25 points): Show your lab instructor or assistant the executing program.**

**EXTRA CREDIT:**

**Part 5: Draw the die in three dimensions**

1. Modify the paint method such that dice are drawn in three dimensions. The "up" face of the die should be mostly showing; a couple of the other faces should also be showing, but in 3-dimensional perspective. This should model an actual die, where the 1 and 6 are on opposite faces of the die (and similarly for 2 and 5, and for 3 and 4). Thus 1 and 6 should never be drawn on adjacent faces.

**checkpoint 5 (10 points): Show your lab instructor or assistant the executing program.**