CIS 201 – Computer Science I
Laboratory Assignment 7

Introduction

In this lab you will:

- Implement a class
- Use a `CompositeSprite`
- Practice using `ArrayList`
- Do timed animation

Create a directory, `Lab7` in your `CS1` directory. Into that directory, download the game `EasyDice.java` from our class directory:

```
/home/student/Classes/201/Labs/Lab07/EasyDice.java
```

Look at the contents of the game. It is functional *except* that it relies on a class, `OneDie`, that does not exist. Your job, in this lab, is to create the `OneDie` class so that `EasyDice` can be run.

Create a new Java source file to implement the class `OneDie`. Insert a header comment for the file with both your lab partners' names. The `OneDie` class should extend the `CompositeSprite` class in exactly the same way that the `EasyDice` class extends the `Game` class. Be sure that your source file is in the same directory as `EasyDice.java`.

Look at the code for `EasyDice`. In your `OneDie` source file, document (in your header block) the signatures all of those methods that are used by `EasyDice` on any object of type `OneDie` in the game. You know that they must all have `public` access rights. (Hint: The class `EasyDice` must be able to call them from outside the class `OneDie`.)

Don't forget the `OneDie` constructor: it will have a signature, too. You can tell what parameters this constructor requires by looking at where the `EasyDice` program creates new instances of `OneDie`. Note that the `this` variable always refers to an object of the same type as the class in which the variable is used. In the `EasyDice` class, the `this` variable is an object of type `EasyDice` but is *also* an object of type `Game` – since an `EasyDice` object *is* a `Game` object because of inheritance. Fo your `OneDie` constructor, then, the constructor's parameter should be a variable of type `Game`.

Checkpoint 1

Show us your `OneDie.java` so far.

Composite sprites

Look at the documentation for `CompositeSprite`. You will see that some of the `CompositeSprite` methods already implement the methods signatures you have documented, so in fact you don't need to implement these! Separate those method signatures that are not implemented in the `CompositeSprite` class from those that are.

Now write stubs for all of the methods that need to be implemented. At this point, you should have code that compiles!

Checkpoint 2

Show us your work at this point, and that both the `EasyDice.java` and `OneDie.java` files compile correctly.

Working with one die

Your `OneDie` object will need fields. You need to keep track of the value shown on the die (an `int`), the game the die is in (this is the thing you pass in the constructor, and it is needed for `randomInt`), and the sprites that are the pips and the white background.

You will need 7 pips; you can use the `setVisibile` method to turn them on or off depending on the face that is showing. The pips will be located as follows:

```
p   p
p p p
p   p
```

A `CompositeSprite` is like the `Game` canvas in that you can add sprites to it. However, unlike the `Game` canvas, the geometry of a `CompositeSprite` is centered at (0,0), and you need to add items to the `CompositeSprite` relative to that center.

In the constructor for your die, you should add eight sprites to it. First add the background as a white rectangle, centered at (0, 0) with size 1.0. The pips are 0.2 by 0.2 and located as shown in the diagram above. You should imagine that the display area of the sprite will have its upper left corner at (-0.5, -0.5) and its lower right corner at (0.5, 0.5). Note that the `addSprite` method in the `CompositeSprite` will add the sprites to the `CompositeSprite`, not to the `Game` canvas. Each of the pips should have its visibility set to `false`.

You constructor will also need to save the `Game`

Now implement the `setValue` method. It should check which number (between 1 and 6) that value is being set to and set the value field *and* turn on the right pips. You probably want to turn all the pips off first and then turn on just the ones you want.

Also implement the `getValue` method to return the current value.

Roll the dice

`roll` takes a time parameter, the amount of time you want the die to roll. It is 2.0 seconds in the `EasyDice` game, but the parameter will determine how long it should roll. When I call `roll`, change the face (randomly...this is why you need the `Game`) and start counting down. Every eighth to quarter of a second, change the face. This means you will be running two different countdown timers, one for the current face and one for the roll. These countdown timers should be declared as `double` field variables.

You will need an `advance` method (like the `move` method in `BallSprite`) so that the timers can be updated. Your logic for the `advance` method will look like this:

```
if (animationTimer > 0) {
  // decrement animationTimer by dT
  // decrement faceTimer by dT
  if (faceTimer < 0) {
    // show new random face and update the die value
    faceTimer = // time to show one face
}
```

Also fix up `isRolling` to return true so long as there is time remaining in the current die roll, in other words, as long as `animationTimer` is positive.

Checkpoint 3

Show us your work at this point. Your `EasyDice` game should now work as advertised!

Two dice!

Modify your `EasyDice.java` file so that it has two dice on the game canvas, one on the left side and another on the right, scaled appropriately. You will need to create a new field variable for the second die – perhaps name the two dice `die1` and `die2`.

In your `advance` method, have the left arrow key roll the left die and the right arrow key roll the right die. Don't print anything in this version.

Checkpoint 4

Show us your work at this point.