Bug-gy Classes

For each of the following exercises, write a new class that extends the Bug class. Override the `act` method to define the new behavior. Then, make a new BugRunner class, in which the `main` method features your `Bug` "relative" in an `ActorWorld`*. Remember that the `Bug`-related objects only do one part of its behavior at a time (e. g., it cannot move several spaces in one call of `act`).

1. Write a class `CircleBug` that is extremely similar to BoxBug, except a `CircleBug` follows a "circular/octagon" path instead of square, if a polygon with a sufficient number of sides can be considered a circle.

2. Write a class `SpiralBug` that drops flowers in a spiral pattern. The spiral's innermost leg must have a side length determined by a parameter, `int length`. You may want to change the world to an `UnboundedGrid` to see the spiral pattern more clearly*. Hint: imitate BoxBug.

3. Write a class ZBug to implement bugs that move in a "Z" pattern, starting in the top left corner and putting down at most one flower per call to the `act` method. After completing one "Z" pattern, ZBug should stop moving. If the ZBug cannot move at any time, it should stop and not attempt to make a new side. Supply the side length of the "Z" as a parameter in the constructor. Have ZBug start making the "Z" facing EAST, even if it is facing another direction. Hint: Use methods from the `Location` class.

4. Write a class `DancingBug` that "dances" by making predetermined number of turns after putting down a side of predetermined length.
    a. The `DancingBug` constructor has a side-length integer and "dance routine" integer array as parameters.
    b. When a `DancingBug` object `acts`, the side-length integer and "dance routine" integer array are used:
        i. A `DancingBug` object puts down a side length, much like a BoxBug.
        ii. Instead of turning 90^{o}, it should turn the number of times given by the current array entry.
        iii. The integer entries in the array represent how many times the bug turns before it moves. For example, an array entry of 5 represents a turn of 225 degrees (recall one turn is 45 degrees).

iv. The following calls of `act` should move the bug forward and put a flower down, one space at a time. Again, the `DancingBug` object puts down a side length's worth of spaces.
v. After that, it should use the next entry in the array to turn the appropriate amount of times, before continuing onto the next side.
vi. Once the entire "dance routine" array is followed, the `DancingBug` object should start over at the beginning of the array, repeating the same pattern.

c. `DancingBugRunner` should create an array and pass it onto the constructer, along with an integer*.

*`SampleBugRunner` class with an unbounded `Grid` and an integer array.

```
import info.gridworld.actor.ActorWorld;
import info.gridworld.actor.Actor;
import info.gridworld.grid.Location;

import info.gridworld.grid.UnboundedGrid;
     // Identifies what an UnboundedGrid is.

import java.awt.Color;

public class SampleBugRunner
{
    public static void main(String[] args)
    {
        ActorWorld world = new ActorWorld(new UnboundedGrid<Actor>());

       // Create an array of integers
       int[] x = new int[2];
        x[0] = 1;
        x[1] = 5;

        // Pass an integer array as a parameter
        SampleBug bob = new SampleBug(x);

        world.add(new Location(20, 20), bob);
        world.show();
    }
}
```

5. Create class called `JumperBug`. The `JumperBug` has the following traits when it `acts`:
    a. If it can, it must jump forward two cells.
    b. It can only "jump" over rocks and flowers.
    c. `JumperBug` never leaves any flowers behind.
    d. If it cannot jump, it `moves` like a `Bug`.

    Before jumping (get it?) straight to the code, consider how it behaves in **ALL** `GridWorld` circumstances. Some of these include:

    a. Being at the edge of the grid.
    b. Interaction with other actors.
    c. What can it land on? What can it move on? What can't it land on or move onto?

6. Create a class called `RandomBug` which can be used to model a population. It has the following rules:
    a. `RandomBug` has a constructor that takes two integer parameters: **`die`** and **`breed`**. Both have values that may range between 0 and 100, inclusively.
    b. Each `RandomBug` object has `die`% chance of dying and `breed`% chance of breeding.
    c. If a `RandomBug` object breeds, then it
        - spawns another `RandomBug` object in a random adjacent, empty square on the grid, meaning that it cannot spawn on a rock or flower,
        - begets `RandomBug` objects of the same chance of breeding and dying as the parent, plus they face the same direction as the parents, and
        - faces the same direction before and after breeding.
    d. If it dies, it will remove itself from the `Grid`.
    e. Every time `RandomBug.act()` is called, a `RandomBug` object can either replicate, die, do both, or do neither.
    f. If it does neither, then it acts like a regular `Bug`.
    g. Hints:
        i. Consider using `Math.random`(). It provides a pseudorandom double (good enough) from 0 to 1.
        ii. You will have to use `Grid` and Location methods. Remember to import `java.util.ArrayList` if you chose to use them.

7. (*Challenge Question*) Create a Bee class, which follows these rules:
    e. If a flower is within a certain integer radius of the bee, rounding down, and that number is less than the smelling radius of a bee (an integer parameter), then the bee "flies" to it, facing the direction towards the eaten flower from its original location.
    f. If there are several flowers within the radius, the bee chooses the closest flower.
    g. If several are of equal distance, the bee chooses one randomly.
    h. If there are no close flowers, the bee will act like a Bug, but it will not leave flowers behind when moving. Consider the implications of this for when the bee reaches a wall.