The purpose of this assignment is to use the `Frog` class from Assignment 1 as the basis for a larger simulation involving red predator frogs and green prey frogs.

The assignment is to be submitted via WebCT before **noon** on **Monday, November 28**. Late assignments will be accepted until noon on November 30, but a 20% penalty will be assessed. Assignments submitted later will *not* be graded. *The full mark for the assignment is 10.*

You have been provided with an incomplete project called `RedGreen`. This includes the familiar classes from the `shapes` project (the `Canvas` class has been slightly modified to enlarge the drawing surface to 500×500 pixels), as well as a modified version of the `Frog` class from Assignment 1, which has been turned into an abstract class. You will write two subclasses of `Frog` called `RedFrog` and `GreenFrog`. You will also complete the `Simulator` class, which is responsible for storing and manipulating red and green frogs as they interact on a grid.

Strict Guidelines:

1. Do *not* modify the `Frog` class or the classes in the `shapes` project.
2. Follow the submission guidelines at the end of this handout.
3. Each of the classes you write/modify must include the usual header.
4. Your classes *must* include properly formatted Javadoc comments. Be sure to test the generation of your documentation.
5. Carefully *re-read* the Course Ethics section on the course web page, especially the remarks concerning **plagiarism**:

   Do not expect that small changes in a program (such as altering comments, changing variable names, or interchanging statements) will avoid detection. If you cannot do the work yourself, it is extremely unlikely that you will succeed in disguising someone else's work.

Class `Frog` (not to be modified)

As noted above, the `Frog` class is modified relative to Assignment 1. The `name` field has been removed, along with the associated accessor and mutator methods. The `jump` method has also been removed. Methods called `frogUp` and `frogDown` have been added — these are the vertical counterparts of `frogLeft` and `frogRight`. In addition, an abstract method called `gridMove` has been added — you will override this in `RedFrog` and `GreenFrog`. Note that `Frog` now has a large number of `public static final` fields (class constants); the names of these should be mostly self-explanatory. Some of the non-static fields have been made `protected`, but `hPos` and `vPos` have been left `private` to ensure that they can only be modified by the methods in `Frog`. (As in Assignment 1, `hPos` and `vPos` represent the horizontal and vertical pixel coordinates of the tip of the frog's head.)

Class `RedFrog`

A red frog is a predator frog that occupies a location on the simulator grid, and only moves diagonally. The class `RedFrog` is a subclass of `Frog`. `RedFrog` has no new fields (of course it inherits the fields of `Frog`), one constructor, and one method called `gridMove` (overriding the abstract method in `Frog`). The constructor takes two `int` parameters — these represent the x- and y-coordinates (in pixels) of the initial position of the frog on the canvas. The constructor should place the frog at this specified position, change the color of the frog's body to `"red"`, change the color of both of its eyes to `"magenta"`, and make the frog visible.

The method `gridMove` takes a single `int` parameter and returns `void`. The parameter will be one of the following four constants found in the `Simulator` class: `NORTH_WEST`, `NORTH_EAST`, `SOUTH_EAST`, `SOUTH_WEST`. You can assume that the frog is currently located at the center of a grid square, and that it is legal to move one step diagonally in the specified direction (the `Simulator` class is responsible for checking all these things, so you don't have to worry about them here). The method should move the frog horizontally *and* vertically as appropriate to effect this diagonal move. All you need to know are the height and width of a grid square — these are stored in the `Simulator` class constants `GRID_SQUARE_HEIGHT` and `GRID_SQUARE_WIDTH`. To get slightly smoother animation, make the frog invisible, move it, and then make it visible again.

Class `GreenFrog`

A green frog is a prey frog that occupies a location on the simulator grid, and only moves horizontally or vertically. The class `GreenFrog` is a subclass of `Frog`. `GreenFrog` has no new fields, one constructor, and one method called `gridMove` (overriding the abstract method in `Frog`). The constructor takes two `int` parameters — these represent the x- and y-coordinates (in pixels) of the initial position of the frog. The constructor should place the frog at this specified position and make the frog visible (there's no need to change the frog's colors).

The method `gridMove` takes a single `int` parameter and returns `void`. The parameter will be one of the following four constants found in the `Simulator` class: `NORTH`, `EAST`, `SOUTH`, `WEST`. You can assume that the frog is currently located at the center of a grid square, and that it is legal to move one step in the specified direction (again, the `Simulator` class is responsible for checking these things). The method should move the frog horizontally or vertically as appropriate. To get slightly smoother animation, make the frog invisible, move it, and then make it visible again.

Class `Simulator`

The `Simulator` class is responsible for placing the red and green frogs on a grid, and for controlling their movements and interactions. The size of the grid is 10×10 (see the class constant `GRID_SIZE`). As you can see, a large number of class constants have been provided for you (including a random number generator named `simRand`). Use these whenever appropriate — you will lose marks if instead you use "magic numbers," i.e., constants hard-coded into your program.

You are required to add the following to the `Simulator class`: four (non-constant) fields, a constructor, and five methods.

`Simulator` Fields

Add two `int` fields named `numRed` and `numGreen` — these will store the number of red and green frogs on the grid, respectively. Also add two fields named `grid` and `nextGrid`. These are two-dimensional arrays of `Frog` references. In brief, `grid` will reference a two-dimensional array that holds the current state of the grid: `grid[i][j]` is either `null` (if grid location (i, j) is empty), or it references a `RedFrog` or a `GreenFrog` object. The field `nextGrid` will be used temporarily during an iteration of the simulation to hold the new grid arrangement; after the iteration is over, `grid` will be set equal to `nextGrid` — more on this below.

`Simulator` Constructor

The `Simulator` constructor takes two `int` parameters representing the number of red frogs and the number of green frogs to create, respectively. For each of these parameters, if it is positive, assign it to the field `numRed` or `numGreen` (as appropriate); if it is negative, instead initialize the field to the constant `DEFAULT_NUM_RED` or `DEFAULT_NUM_GREEN` (as appropriate). The constructor should call the pre-written method `drawPips`, which places small "pips" at the corners of the grid squares. It should then construct the two-dimensional array referenced by `grid`, and call the method `createFrogs` to populate the grid with frogs.

`Simulator` Methods

- `createFrogs` — a `private void` method with no parameters. This method constructs `numRed RedFrog` objects and `numGreen GreenFrog` objects. Each frog should be placed at a random location in the grid. Use the random number generator to a generate random row and column grid indices, and then use a formula to determine the corresponding horizontal and vertical pixel coordinates of the center of this grid square — these pixel coordinates are what you need to pass to the frog's constructor. Note that two frogs cannot occupy the same grid location, so for each frog you need to repeatedly generate a random grid location until you hit one that is empty.

- `runOneStep` — a `public void` method that takes no parameters. This method runs one iteration of the simulation as follows. First it constructs a new two-dimensional array referenced by `nextGrid`. It then scans through every location in `grid`, and for each green frog that it finds (use `instanceof`), it calls `processGreen`, passing it the row and column indices of the green frog. It then repeats this process for the red frogs, scanning through every location in `grid`, and for each red frog that it finds, calling `processRed`, passing it the row and column indices of the red frog. (*Note: it is very important that the green frogs be processed before the red frogs.*) When finished, it sets `grid` equal to `nextGrid` (these arrays will have been modified by `processGreen` and `processRed`).

  It is important to understand how frogs are moved by `processGreen` and `processRed`, and how the two-dimensional arrays `grid` and `nextGrid` are used by these two methods. During an iteration, `grid` holds references to frogs that *have not yet moved,* while `nextGrid` holds references to frogs that have already moved. When a green frog tries to move to a new location, you must check that this new location is not occupied by a frog that has not yet moved *nor* by a frog that has already moved. In other words, the location to which the green frog wants to move must be empty (`null`) both in `grid` and in `nextGrid`. The situation is a

little different for red frogs. Since the green frogs move first, by the time a red frog tries to move, there are no green frogs referenced by `grid`. The location to which a red frog wants to move must be empty in `grid` (else it will land on top of a red frog that has not yet moved), and must either be empty in `nextGrid` *or* must contain a green frog in `nextGrid` (in which case the red frog eats the green frog). Any remaining details are given in the descriptions of `processGreen` and `processRed` below.

- `runMultipleSteps` — a `public void` method that takes a single `int` parameter specifying the number of simulator iterations. Simply call `runOneStep` this number of times.

- `processGreen` — a `private void` method that takes two `int` parameters representing the row and column indices of a green frog on the grid. This method generates a non-negative random number that is less than `NUM_DIRECTIONS`, and uses this to determine in which of four directions — north, south, east, west — the frog should try to move (use the constants `NORTH`, `SOUTH`, etc.). Whatever direction is chosen, the frog will try to move *one* grid location in that direction. If the new location is on the grid and if frog won't land on top of another frog (see above), make the new location in `nextGrid` reference the frog, set the current location in `grid` to `null`, and move the graphical representation of the frog on the canvas by calling `gridMove` and passing it the integer representing the direction that was chosen. If the new location is off the grid, or if the new location is already occupied, the frog will not move, so make the current location in `nextGrid` reference the frog, and set the current location in `grid` to `null`.

- `processRed` — a `private void` method that takes two `int` parameters representing the row and column indices of a red frog on the grid. The red frog scans each of the four locations that are its diagonal neighbors, beginning northwest and proceeding clockwise. If none of these neighboring locations is occupied by a green frog, then the red frog doesn't move, so make the current location in `nextGrid` reference the frog, and set the current location in `grid` to `null`. If at least one of these locations *is* occupied by a green frog, then the red frog will eat the first green frog it encounters in its scan. To make a red frog eat a green frog, first make the green frog invisible, then decrement `numGreen`, make the new location in `nextGrid` reference the red frog, set the current location in `grid` to `null`, and move the graphical representation of the red frog on the canvas by calling `gridMove` and passing it the integer representing the direction that was chosen (use the constants `NORTH_WEST`, `NORTH_EAST`, etc.).

Submitting Your Assignment

You should place your results in a zipped file called `usrnmA2.zip` (where `usrnm` is your Mount Allison username). This file should contain `RedFrog.java`, `GreenFrog.java`, and `Simulator.java`. Submit the zipped file to WebCT.

Don't forget to include proper headers and Javadoc-style comments.