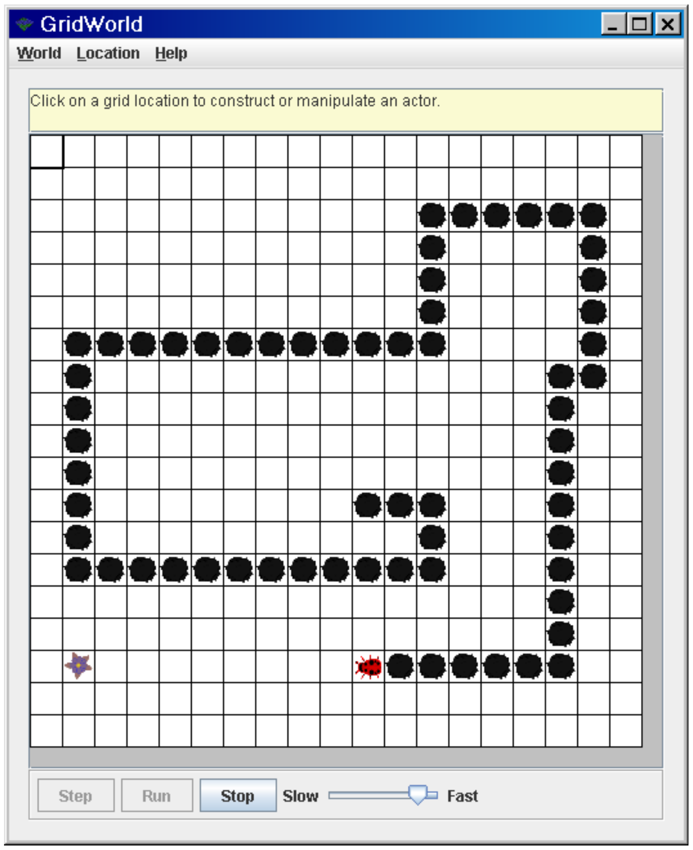
- Classic Snake Game Project -
- - GridWorld Case Study - -

- Dave Ruth - - AP Computer Science -
- - - Niles North High School - - -



Click on a grid location to construct or manipulate an actor.

This is a project that will allow students to clone the classic **Snake** Game (sometimes known as **Nibbles**). Students should have successfully completed at least parts 1, 2 and 3 of the GridWorld Case Study Manual and the respective question sets. They should also be familiar with using ArrayList<E>. A SnakeGame class with a complete main method and will compile and execute (immediately after the empty SnakeBug class is written .. see Question 0). This will allow students to begin with a working GUI with KeyEvent handling.

Students will have to write a class SnakeBug that extends the Bug class. They will write a constructor and three methods, as well as override the act() method from Bug. The instructions for writing the SnakeBug class are phrased in the form of 'AP Test type' free response questions. The amount of code to be written (about 60 to 70 lines of code not including imports) should be equivalent to approximately 1½ to 2 free response questions. It is recommended that students work on the questions in the order they appear.

Question 0. class SnakeBug
Create a class called `SnakeBug` that extends class `Bug` in a new source file. Copy the imports that are in `SnakeGame` and then create two instance variables:

```
private int grow;//current number of pieces the snake should grow
private ArrayList<Location>snake;//list of Locations of the snake body
```

Next, write a default constructor (no arguments) that initializes variable `grow` to what will be the initial length of the snake (4 is a good value) and `snake` to an empty `ArrayList<Location>();`

Now, run `SnakeGame`.

Question 1. randomFlower method of class SnakeBug
`public void randomFlower()` method has no parameters and will create a `new Flower` object in a random, empty `Location` in the `Grid`. You may use a local variable `Grid<Actor> gr` as a reference to the `Grid` the `SnakeBug` is in. Select a random cell from `gr` that is empty (i.e. `null`). This will require a loop. Make no presumptions about the size of `gr`. Use `Grid` methods of to determine its number of rows and columns for your loop. Once found, create a `new Location` with the row and column value for the random cell and 'put' a `new Flower` object into `gr` at that newly determined `Location`. Uncomment the method call to `head.randomFlower()` in the `SnakeGame` `main` method. Run `SnakeGame`. One `Flower` should appear in a random `Location` in the `Grid`.

Question 2. addRockBehindHead method of class SnakeBug
This void method has a `Location loc` parameter. It creates a `new Rock` Object and adds it to the `Grid` the `SnakeBug` is in at `Location loc` . Also, add `loc` to the beginning of `ArrayList snake`.

Question 3. eraseTail method of class SnakeBug
The last element of `ArrayList snake` will be the `Location` of the last `Rock` in the snake. Remove this `Rock` from `gr` at this `Location` and remove the last element of `ArrayList snake` .

Question 4. override the act() method SnakeBug class
Override method act() so the `SnakeBug` acts with the following behavior. If the `SnakeBug` cannot move, simply end the game. (*please note: this is one of the lamest GAME OVER's in the history of Video Games*)

```
  javax.swing.JOptionPane.showMessageDialog(
      null, "Score: " + snake.size(), "GAME OVER!", 0);
```

Otherwise, check if a `Flower` Object is directly ahead of the `SnakeBug`. It is recommended you make local variables `Grid<Actor> gr` and `Actor inFront` which is in the `Location` immediately in front of the `SnakeBug` based on its direction.

If the `Actor inFront` is a `Flower` Object … i.e. `if(inFront instanceof Flower)`
then increase variable `grow` by 3, remove the `Flower` from the `Grid`, and spawn a `new Flower` by calling the `randomFlower` method.

Next, the `SnakeBug` should move like a `Bug` and add a `new Rock` Object at the `Location` it just moved from… i.e. `addRockBehindHead`

Also, if variable `grow` is positive, simply decrease its value by 1, otherwise erase the `snake` tail.

```java
//SnakeGame.java
import info.gridworld.actor.*;
import info.gridworld.grid.*;
import java.util.ArrayList;

/**
* This is a clone of the classic Snake game (a.k.a. Nibbles).
* The GUI is the standard for GridWorld and the snake
* is controlled by the arrow keys on the keyboard.
*  (the code is provided for KeyEvent handling in the main)
*/
public class SnakeGame {

 /**
  * ActorWorld variable used for GUI...
  * Initialized to 19x19 Grid for aesthetic viewing.
  */
  public static ActorWorld world = new ActorWorld(new BoundedGrid(19, 19));

 /**
  * The only Actor that actually moves in the game.
  * Its direction is controlled by the keyboard arrow keys.
  */
  public static SnakeBug head = new SnakeBug();

  /**
  * Handles game simulation with one bug whose moves
  * NOTE: YOU DO NOT NEED TO WRITE ANY CODE IN THE main
  */
  public static void main(String[] args) {

    head.setDirection(0); //facing North
    world.add(new Location(17, 9), head); //initial Location of the Snake

    //head.randomFlower(); //<==UNCOMMENT AFTER randomFlower() HAS BEEN WRITTEN!

    java.awt.KeyboardFocusManager.getCurrentKeyboardFocusManager()
        .addKeyEventDispatcher(new java.awt.KeyEventDispatcher() {
          public boolean dispatchKeyEvent(java.awt.event.KeyEvent event) {
            String key = javax.swing.KeyStroke.getKeyStrokeForEvent(event).toString();
            if (key.equals("pressed UP"))
              head.setDirection(0);
            if (key.equals("pressed RIGHT"))
              head.setDirection(90);
            if (key.equals("pressed DOWN"))
              head.setDirection(180);
            if (key.equals("pressed LEFT"))
              head.setDirection(270);
            return true;
          }
    });
    world.show();
  }
}
```

Some possible additions to the game (perhaps extra credit).

#. Fix the bug (*no pun intended*) where some keystrokes appear to be skipped, especially at a slow speed. You may notice that if two arrow keys are pressed within a given frame, the second key sets the direction of the `SnakeBug` for the next move before it had a chance to move in the direction from the first key.

Here is a quick fix. Create a queue of directions in `SnakeBug`

```
    public ArrayList<Integer> directionQ;
```

It's O.K. if this is the first time students are introduced to the 'First In, First Out' data structure.
Now, replace `head.setDirection(0)` with `head.directionQ.add(0)` in `SnakeGame` and similarly with the other three directions.

All that is left is to write an `if` statement at the beginning of the `act()` method. If the direction queue is not empty, set the direction of the `SnakeBug` to the value of the first element of the queue and then remove that first element of the queue.

##. Make the game more challenging. Add an instance variable to class `SnakeBug` called `framesToRandomFlower` and set it to 15 or so. Decrement it each time the `SnakeBug` moves. If it reaches 0, remove the current `Flower` and create a new `Flower` and set `framesToRandomFlower` back to 15. If the `SnakeBug` reaches the `Flower` within 15 moves, set `framesToRandomFlower` back to 15. This requires the player to make more risky choices to reach the `Flower` within 15 moves. What's neat is the built in `Flower Color` change (i.e. 'wilting') will give the player a visual cue as to when the `Flower` will relocate itself.