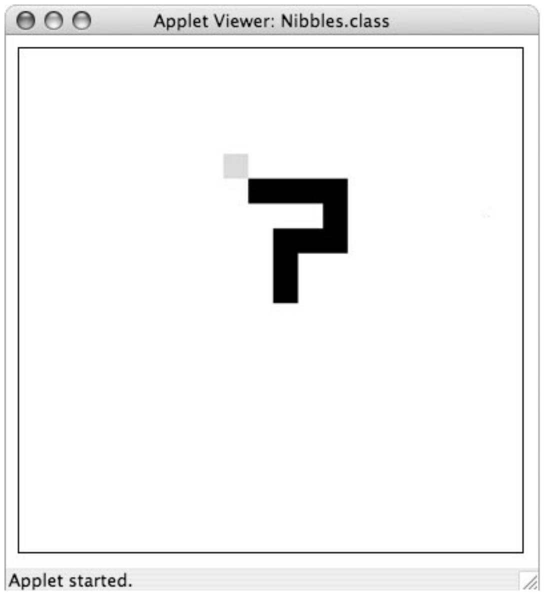
CSIS10A Final Project
Nibbles

Objective: To gain experience working with 1 dimensional arrays.

The Problem: Nibbles is a snake. Nibbles moves around a field, looking for apples. Unfortunately, Nibbles is not a very clever snake. It will try to eat anything it can reach. When it eats an apple, it grows. When it eats the electric fence around its field, it dies. When it gets all twisted up, it can even try to eat itself, which also causes it to die. The player of the game will control the movement of the snake. The objective, of course, it to eat as much food (and grow) as much as possible, without dying.

The Game Below is a picture of Nibbles. In the picture, Nibbles is the winding black ribbon that looks a bit like the top of a question mark. The small gray rectangle is the apple. A user controls the movement of the snake with the arrow keys on the keyboard. ***Important: You must click on the window displaying the snake before your program will respond to the arrow keys!*** The snake constantly moves by extending its "head" in the direction indicated by the last arrow key the user presses. While it is hard to tell from the picture, when the picture was taken, the snake's head was the square just below and to the right of the apple. It was moving to the left of the screen. The square at the bottom of the question mark is the end of the snake's tail. Normally, each time the head moves into a new cell, the last cell of the snake's tail is removed from the screen. If the snake manages to eat the apple, it becomes a few squares longer. It does not grow all at once. Instead, for a few steps after it eats, the snake's head is allowed to move into new squares while none of the squares in the tail are removed, thus simulating the growth of the tail. Note that the snake can go straight or it can turn, but it cannot reverse its direction with a single click of an arrow key.



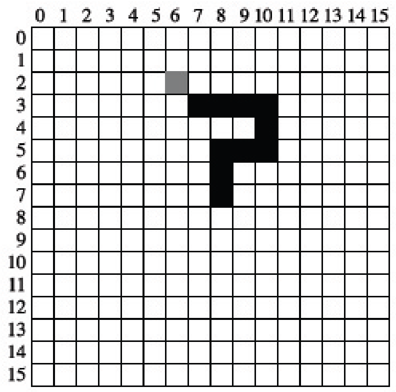
Design The Nibbles program consists of 3 classe definitions:

- Nibbles - the window controller that initializes the display and handles the user input,
- Snake - an ActiveObject extension that animates the snake as it slithers around the screen,
- Direction - a class whose values are used to represent the four directions in which the snake can move: UP, DOWN, LEFT and RIGHT.

You will be given the working versions of the Nibbles and the Direction class, and you will write code for the Snake class.

The Nibbles Class: The Nibbles class extends WindowController. It begins by creating the apple (food), and the snake. It then receives the clicks of the arrow buttons and tells the snake to change direction accordingly. The Nibbles file you have been given is already working. DO NOT CHANGE THE NIBBLES CLASS.

The Snake Class: This class manages an array of FilledRects called body[] that show the current position of the snake. The FilledRects in the snake body are restricted to fall only on allowable squares in the field on which the snake moves. We could have used a 2D array to represent this field, but it is simpler to restrict the FilledRect positions so they fall only at locations determined by a math formula. For example, the field below is divided up into a grid like a checkerboard. While the boundaries between the cells of this grid are not visible on the screen as the game is played, the cells determine where it is possible to put the apple and the pieces of the snake's body. The picture below shows how the image of the snake and the apple (shown above) might look if the grid in which the game is played was visible.



If we decide that a CELLSIZE is to be set to 10 (meaning 10 pixels by 10 pixels), then we can identify the x and y positions for the different pieces in the figure above by multiplying the row and column numbers by 10. In this case, the "head" of the snake (nearest the "apple") would be located at 70,30 (because it is in column 7, row 3). If the Snake were to continue moving left, we would just need to create a new FilledRect at location 60,30 (of size 10x10), and erase the FilledRect at the snakes tail (the square at Location 80,70).

The Direction Class: The class Direction is used to encode the direction in which the snake is currently moving. There are four possibilities: Direction.UP, Direction.DOWN, Direction.LEFT and Direction.RIGHT. Internally, the representation of a Direction is similar to that used for Locations. A Direction is just a pair of values indicating how the snake's head should move in the horizontal and vertical directions. Each of these two values can be either 0, 1 or -1. To convert these numbers into location offsets, we just need to multiply them by CELLSIZE. In this way, our snake body parts will fall on the grid above, without needing to declare/define every FilledRect in a 2D array.

In addition, the Direction class provides several methods that can be used to manipulate Direction values themselves:

```
public int getXchange();
public int getYchange();
public boolean isOpposite( Direction newDir );
```

The **getXchange** and **getYchange** methods return the amount of horizontal or vertical motion (either 0, 1 or -1) associated with a Direction value. The **isOpposite** method returns true if the Direction passed as a parameter is the opposite of the direction on which the method is invoked.
Like the Nibbles class, the Direction class is already working. DO NOT MODIFY THE DIRECTION CLASS file.

The Snake Class: The most interesting class is the Snake. The Snake is an active object. It continuously moves the snake around the screen checking at each step whether the snake has found food and should grow or has made an illegal move (out of bounds or into its own body) and should die.

If the user types an arrow key, the Nibbles controller gets the input and calls the setDirection method of the snake

```
public void setDirection (Direction newDirection);
```

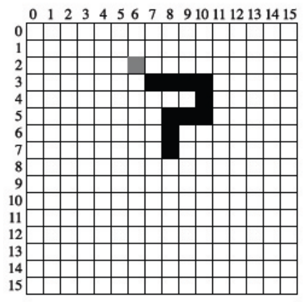
to inform the snake that the direction in which it moves should be changed. The snake should ignore the controller if the new direction is the opposite of the snake's current direction.

The snake moves and tries to eat inside its run method. Between each move, it pauses for a while so that it slithers at a realistic speed. The snake normally moves by stretching its head forward one cell and then removing the last piece of its body from the screen.

We will implement this motion using two separate methods: one to **stretch** the snake by adding a cell next to its current head and the other to **shrink** the snake by removing its tail. This will make it easy to make the snake grow after it finds the apple. After the snake finds the apple, you can make it grow by letting it take several steps in which it stretches but does not invoke your "shrink" method. Similarly, when the snake should die you should make it gradually disappear from the screen by repeatedly invoking your "shrink" method without calling "stretch".

To implement "stretch" and "shrink", you will keep track of the FilledRects of the parts of the snake's body in an array. In addition, you will need an **int** instance variable to keep track of the snake's current length. Since the snake's length changes as the game is played, you will have to create an array large enough to hold as large a snake as you can imagine any player will ever manage to produce. Make sure to check that the snake never becomes larger than the array.

Each element of the array in the Snake class identifies the FilledRect of one portion of the snake's body. For example, consider the snake shown in the pictures of the game field shown above and repeated here:



Since this snake occupies 10 cells of the field, 10 elements in a FilledRect array would be used to describe it. The picture below suggests what the array would look like. As a shorthand in this figure, we have written pairs like (70,30) and (100,50) to represent Location values, but you should realize that you cannot actually do this in your code. Instead of (70,30) you would have to say new FilledRect(70,30,10,10,canvas) (where 10,10 is the width and height of the rectangle).

The snake in the example we have been consiering, would then be described by an array with the following FilledRect Locations:

Body[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
(70,30)	(80,30)	(90,30)	(100,30)	(100,40)	(100,50)	(90,50)	(80,50)	(80,60)	(80,70)

The FilledRect at the front end of the snake is at Location 70,30, this FilledRect is stored in the 0th element of the array. The piece next to this piece is at Location 80,30 and it is shown in the 1st element of the array. The locations of the remaining pieces of the snake's body are placed in the remaining cells of the array in consecutive order.

You cannot tell from either the picture of the game grid or the FilledRect array which end of the snake is the head and which end is the tail. This is because the pieces of the snake can be stored in the array in either order as long as you remember which order you decided to use while writing the code for your Snake class.

If you decide to put the head in the 0th element of the array (recommended) and the tail in the last used element of the array, then the code you write to "stretch" the snake will have to include a loop to move all the elements in the array over by one position to make room for the new head. On the other hand, you can shrink the snake very easily by just removing the last element in the array.

If you decide to put the tail in the 0th element and the head in the last element, then the code to stretch the snake will be simple, but the code to shrink the snake will require moving all the elements of the array down one slot to fill in the vacancy left when the old tail is removed. Either way, make sure that you check to make sure that the array is not full before adding a new piece to the snake's body. If it is full, you can either simply continue the game but not let the snake grow any more or stop the game declaring that the player has won.

To move the snake, your code should first determine the Location that will anchor the FilledRect for the new head based on the Direction in which the snake is moving. It should check if the Location it is going to move into is out of the bounds or if it is occupied by part of its own body. If either of these conditions apply, the snake dies. Next, your code should check to see if the apple is at the Location that will become the new head. If so, the snake should eat the apple and grow by one cell on each of the next 4 moves it makes. The snake eats the apple by signalling for itself to grow by 3 more links and moving the apple position to a new random location

As we have suggested above, your snake should grow by being allowed to make several moves in which the snake stretches but does not shrink. Thus, your run method will contain one main loop. Each time around the loop you will pick a new Location and "stretch" the snake to move into this new position. Most times around the loop you will also shrink the snake by removing its last cell. For several iterations after eating the apple, however, your snake will skip the "shrink" step in the loop.

Finally, when the snake dies you should make it wither away rather than simply stopping the program. You can do this by writing a separate, simple loop that repeatedly shrinks the snake and pauses.

<u>Your Job</u> You only need to write code in the Snake class for this program to work:

Part I. For the first part, we will provide all the classes for you except the Snake class. Your job for Part I is to complete the snake constructor.

The snake constructor is passed the number of cells in one side of the playing area (established as NCELLS in Nibbles) and the size of the cell FilledRectangles (defined as CELLSIZE in Nibbles). This allows the snake to position itself properly at the center of the field and create FilledRects of the appropriate size. Having constants to store the NCELLS and CELLSIZE information means we can change the settings very easiy to increase the number of cells and decress the cellsize, or vice versa. The constructor should save these first two parameters as instance variables (they are not constants to theSnake).

So the snake can determine if it has eaten the apple, the snake constructor also needs to receive the (FilledRect) apple that was passed from Nibbles, which should also be stored in an instance variable. Similarly the snake needs to remember the border established in Nibbles as well and the canvas from Nibbles (type DrawingCanvas).

After initializing variables, the constructor should then initialize the body array to the largest possible snake size you can imagine playing. It should set the first cell of body (cell 0) to a (GREEN!) FilledRect of the appropriate size at the center of the screen, and set the toGrow variable to 3 so that 3 more links will be added as the snake begins moving. Also set the currentLength to one, and initialize the randomCell object to be able to generate numbers between 0 and NCELLS-1 (for moving the apple). The last thing the constructor needs to do is invoke the start() method to kick off the active object process.

Test your constructor. You should see a green square in the center of the playing area. It won't move yet.

Part II. Examine the **run**() method. It is partially completed with an outer while loop that repeats forever, so every time the snake dies it is reincarnated for a new round. The inner loop repeats while currentLength is > 0 (this is to get you started. It should be changed to repeat as long as the snake doesn't eat itself or go out of bounds, but you can address this later.) The inner while loop in the run method repeatedly invokes **stretch** and **shrink** to make the snake move. These methods need to be defined using the comments placed in them as a guide.

When these two methods are complete, test again. The snake should move and change direction, but it won't grow when it hits the apple yet nor will it die at the boundary edge or when it crosses itself.

Part III. To make the snake grow, add an if statement in the run method after stretch() is invoked. This if statement shold check if body[0].overlaps(apple) and if so, set toGrow = GROWTHRATE and move the apple to a new randomCell (converted to a Location by multiplying by cellsize). Another if statement in front of shrink() will avoid shrinking the snake if toGrow is bigger than 0. Otherwise it will shrink the snake. **TEST**.

Part IV. When you are ready for the snake to be able to die, define the methods outOfBounds() and ateSelf() in the snake class. These methods take no parameters but just returns true if the location of body[0] is out of bounds, or overlaps with any other link in the array. Then, modify the while loop in run() that checks currentSize and use a logical expression combining such as !outOfBounds() and/or !ateSelf() to determine if the loop should continue. Finally, you can allow the snake to reincarnate by re-initializing body[0] at the end of the run method where indicated.

<u>Submitting YourWork</u> Before turning in your work, be sure to double check both its logical organization and your style of presentation. Make your code as clear as possible and include appropriate comments describing major sections of code and declarations. PLEASE INDENT YOUR LOOPS AND CONDITIONALS properly!!