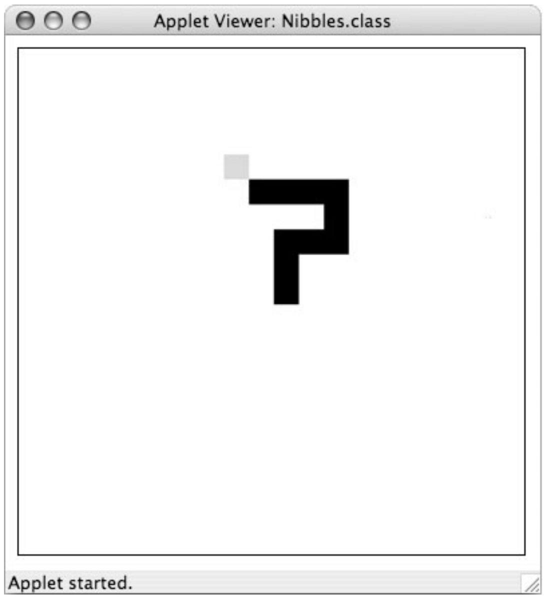
Programming Exercise
Nibbles

Objective: To gain experience working with 2 dimensional arrays.

The Problem Nibbles is a snake. Nibbles moves around a field, looking for food. Unfortunately, Nibbles is not a very clever snake. It will try to eat anything it can reach. When it eats food, 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 food. 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 food. 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 food, 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 5 classes:

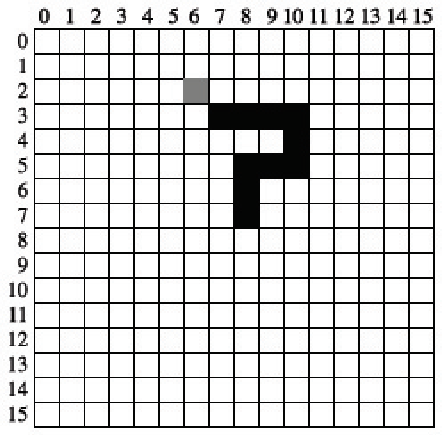
- `Snake` - an ActiveObject extension that animates the snake as it slithers around the screen,
- `NibbleField` - a class used to represent the 2D grid that the snake lives within,
- `Nibbles` - the window controller that initializes the display and handles the user input,
- `Position` - a simple class that is used to represent positions within the NibbleField, and

- `Direction` - a class whose values are used to represent the four directions in which the snake can move: UP, DOWN, LEFT and RIGHT.

Initially, we will give you working code for all but the `Snake` class. We will ask you to implement the snake using our implementations of the other four classes. Then, we will ask you to switch to a version of the `NibbleField` class that is not complete and add the missing pieces needed to make it work with the other code.

The `Nibbles` Class: The `Nibbles` class extends `WindowController`. It begins by creating the field and the snake. It then receives the clicks of the arrow buttons and tells the snake to change direction accordingly.

The `Position` Class: The field on which the snake moves 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 food and the pieces of the snake's body. The picture below shows how the image of the snake and the food (shown above) might look if the grid in which the game is played was visible.



Each cell in this array can be precisely identified by two numbers indicating the row and column in which the cell is located. For example, in the picture the food is located in row 2 and column 6, and the head of the snake is found at row 3 and column 7. We can write these positions as (2,6) and (3,7). The `Position` class is used to combine such a pair of grid coordinates into a single object much as an object of the `Location` class encapsulates the x and y coordinates of a point on the canvas as a single object. Unlike canvas coordinates, the row and column numbers identifying a cell in our Nibbles grid must be integers. Also, the convention is to write the number that describes the vertical position of a cell before the number that describes its horizontal position. This is the opposite of the convention used with canvas coordinates.

The `Position` constructor takes a row and a column as parameters:

```
public Position (int row, int col)
```

In addition to this constructor, the `Position` class provides the following methods:

```
public int getRow();
public int getCol();
public boolean equals(Position otherPos);
public Position translate(Direction dir);
```

The individual row and column values can be extracted from a `Position` using the `getRow` and `getCol` accessor methods. The boolean `equals` method checks if two `Position`s are equivalent. The `translate` method is explained shortly in the discussion of the `Direction` class.

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 far the snake's head should move in the horizontal and vertical directions. Each of these two values can be either 0, 1 or -1.

The most important method used with `Direction` values is actually associated with the `Position` class rather than with the `Direction` class.

```
public Position translate(Direction curDirection);
```

Given a `Position` named `curPos` and a `Direction` named `curDir`, an invocation of the form:

```
curPos.translate(curDir)
```

will return a new position obtained by moving one step from `curPos` in the direction described by `curDir`.

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.

The `NibbleField` Class: The `NibbleField` represents the actual contents of the game grid. Most of the necessary information is encoded using a two-dimensional array containing one entry for every cell in the grid. The entries of the array hold `FilledRect`s. If an entry in the array is null, it means there is nothing in the corresponding cell. If it is not null, it refers to the `FilledRect` drawn on the screen to represent the food or the part of the snake that occupies the corresponding cell in the game grid.

The only parameter expected by the constructor for a `NibbleField` is the canvas.

```
public NibbleField(DrawingCanvas canvas);
```

The constructor uses the `getWidth` and `getHeight` methods of the canvas to decide how big the game grid should be. It also places a piece of food at a random location within the field.

The `Snake` will interact with the `NibbleField` using several methods:

```
public void addItem(Position pos);
public void removeItem(Position pos);
public boolean outOfBounds(Position pos);
public boolean containsSnake(Position pos);
public boolean containsFood(Position pos);
public void consumeFood();
public Position getCenter();
```

The snake can ask the `NibbleField` to place a piece of its body (probably its head) in a given cell by invoking the `addItem` method. It can also remove a part of its body (usually its tail) from the screen by invoking `removeItem`.

Before moving, the snake can use the `outOfBounds` method to ask the field if a particular position is out of bounds. The snake can determine whether a given position in the field contains food or some part of the snake's body using the `containsFood` and `containsSnake` methods.

When the snake gets lucky enough to eat the food, it can tell the field to remove the food and place a new piece of food somewhere else on the field by invoking the `consumeFood` method. This method should be called before moving part of the snake into the cell that had contained the food.

Finally, rather than starting its life in some random position, the snake likes to start in the center of the grid. To do this it has to ask the field where the center is using `getCenter`.

The `Snake` Class: The most interesting class is the `Snake`. The `Snake` is an active object. It continuously moves the snake around the field 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);
```

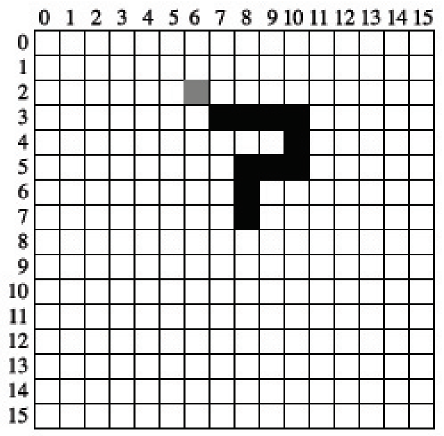
to inform the snake that the direction in which it moves should be changed. The snake should ignore the controller if the new direction is the opposite of the snake's current direction.

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.

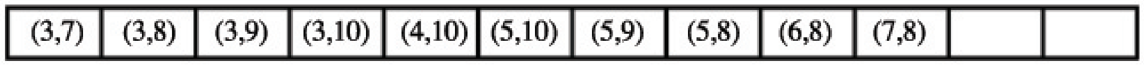
We urge you to implement this motion using two separate methods: one to stretch the snake by adding a cell next to its current head and the other to shrink the snake by removing its tail. This will make it easy to make the snake grow after it finds some food. After the snake finds food, you can make 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 `Position`s 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 `Position` 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 below:



Since this snake occupies 10 cells of the field, 10 elements in a `Position` 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 (3,7) and (5,10) to represent `Position` values, but you should realize that you cannot actually do this in your code. Instead of (3,7) you would have to say `new Position(3,7)`.



The cell at one end of the snake is in row 3, column 7 and this `Position` is stored in the 0th element of the array. The piece next to this piece is at row 3, column 8 and its `Position` is shown in the 1st element of the array. The positions 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 `Position` 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 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 cell that will become the new head based on the `Direction` in which the snake is moving. It should check if the cell 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 food is in the cell that will become the new head. If so, the snake should eat the food and grow by one cell on each of the next 4 moves it makes. The snake eats the food using the `consumeFood` method of the `NibbleField`, which also generates a new piece of food somewhere else in the field.

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 position 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 food, 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.

Your Job You will implement two of the classes that make up the Nibbles program:

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 write a complete version of the `Snake` class.

The snake constructor is passed the `NibblesField` that it wanders through as its only parameter. The constructor should create and initialize the array to hold the location of its body. It should use the field's `getCenter` method to decide where to place the snake initially. The snake should begin moving UP from this position. During the first few steps it takes, the snake should be allowed to grow to its initial length (3). You are responsible for defining a `run` method, a `stretch` method, a `shrink` method, as well as any additional methods you need to make the snake behave as described. You should not need to modify any other classes.

Part II. Once you have a working `Snake` class, you should replace the complete version of `NibblesField` we provided for part I with the incomplete version we want you to use for part II.

Your task for Part II is to complete the `NibblesField` class by filling in the bodies of four of its methods: `removeContents`, `showContents`, `isOccupied`, and `outOfBounds`.

Implementation Here are some hints on how to approach the problem of implementing the snake.

Start by constructing a snake that is only 1 cell long. Write a loop in the `run` method that will make this snake move in a straight line by calling grow and shrink. As the snake tries to move out of the bounds of the game, have the snake detect the boundary and die. The snake should die by falling out of the main loop of the run method.

Once that is working, use the values passed in the `setDirection` method in response to arrow clicks to cause the snake to change direction. At this point, your program should work correctly until the snake runs into the food (at which point it will encounter an error in our `NibbleField` class).

Now, get the snake to eat the food. Change the loop that moves the snake so that after the snake eats the food, the snake will take several steps in which it stretches but skips the call to shrink. You will need a counter to keep track of how many steps the snake has taken since it ate to do this.

Now that your snake can grow to be longer than one cell, it has to worry about running into itself. Add code to make the snake die if it runs into itself. Add a short loop after the main loop that slowly removes all the pieces of the snake from the screen when it dies.

Once your `Snake` class is complete, move on to Part II as described above.

Submitting Your Work 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.