Lab 9

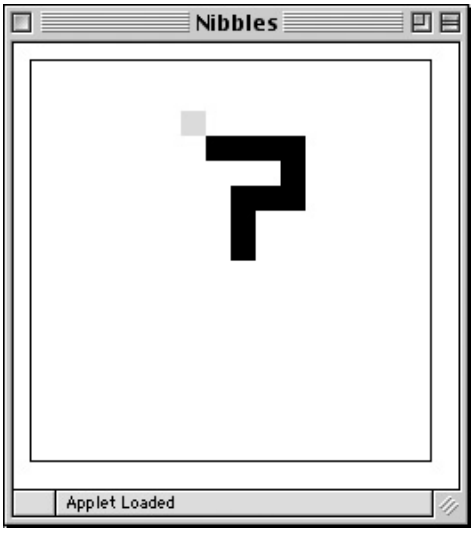
Nibbles

Objective. To gain experience using one-dimensional and two-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.

See the handouts page to download a copy of the starter code and for a working version of the Nibbles game. *You must click on the window displaying the snake before the program will respond to the arrow keys!*

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. 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 be low 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 tap 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.

This lab is divided into two parts:

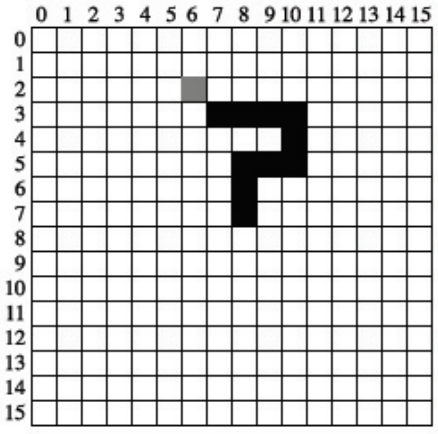
- **Part I.** You will write the `Snake` class. For this part, we provide working code for all but the `Snake` class. You should implement the snake using our implementations of the other four classes.

  Note: when you open the starter project, you will not see the `NibbleField` *class listed with the other classes. We have included this class in a library that does not appear in the BlueJ window. You will, however, see a* `NibbleFieldPart2` *class, which you will use in the second part of the lab...*

- **Part II.** After your snake is working, you will implement your own `NibbleField` class. We will discuss below how to change your project so BlueJ uses a version of `NibbleField` defined by you instead of the one we give you in the library.

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 column 6 and row 2, and the head of the snake is found at column 7 and row 3. We can write these positions as (6,2) and (7,3). 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 column and row numbers identifying a cell in our Nibbles grid must be integers.

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

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

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

```
public int getCol();
public int getRow();
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 `Location`s. 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);
```

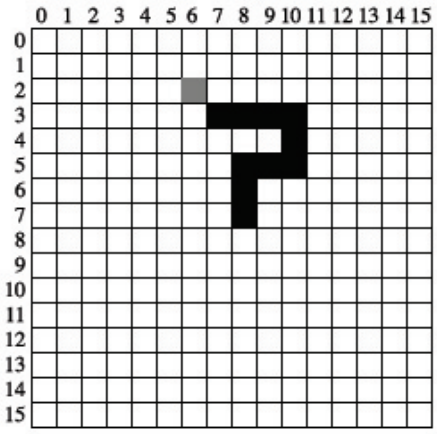
to inform the snake that the direction in which it moves should be changed. The snake should ignore the controller if the new direction is the opposite of the snake's current direction.

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

(7,3)	(8,3)	(9,3)	(10,3)	(10,4)	(10,5)	(9.5)	(8,5)	(8,6)	(8,7)

The cell at one end of the snake is in column 7, row 3 and this `Position` is stored in the 0th element of the array. The piece next to this piece is at column 8, row 3 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.

Implementation Outline

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. Remember the `NibbleField` class is pre-loaded, so it doesn't appear initially in the project. 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 begin working on your version of the `NibbleField` class. To do this:

- Open the Lab9Nibbles project folder in the Finder. Inside, you will find a folder entitled "+lib". It contains the `NibbleField` implementation we provided for you to use while writing the Snake. Change the name of the folder to "part1+lib".
- Return to BlueJ, and Open the `NibbleFieldPart2` class in your project.
- Replace the class declaration:

  ```
  public class NibbleFieldPart2 {
  ```

  with

  ```
  public class NibbleField {
  ```

- Also, change the name of the constructor from `NibbleFieldPart2` to `NibbleField`.

This version of `NibbleField` in this file is identical to out library version, except that it is missing four methods. Your task for Part II is to complete the `NibblesField` class by filling in the bodies of the following methods: `removeContents`, `showContents`, `isOccupied`, and `outOfBounds`.

Design Preparation

This week we will again require that you prepare a written design for your program before lab. As always, we will briefly examine each of your designs to make sure you are on the right track. Our design template is on the handouts page, if you wish to use it. You will also find it useful to refer to the starter code listings at the end of this handout.

Keep in mind that a good design includes the following items for each class:

- A list of the instance variables you expect to include in the class definition;
- the header of the constructor for the class (including all parameter declarations);
- the headers of the methods you expect to define (including all parameters) in the class and a brief description of the function of the method;
- a sketch of the code used in the body of each method and constructor.

If you do not do a good design, you are likely to make very little progress on this program during the lab. Be prepared so that you can take advantage of the lab period.

Implementation Hints

Here are some hints on how to approach the problem of implementing the snake, *once you have completed the design*.

- 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.

After finishing the snake, write your own version of `NibbleField`.

Submitting Your Work

Once you have saved your work in BlueJ, please perform the following steps to submit your assignment. Don't forget to change the name of your folder to include your last name.

- First, return to the Finder.
- Click "Go" and select "Connect to Server."
- For the server address, type in "Cortland" and click "Connect."
- Select the button next to "Guest" and click "Connect."
- A selection box should appear. Select "Courses" and click "Ok."
- You should now see a Finder window with a "cs134" folder. Open this folder.
- You should now see the drop-off folders for the three labs sections. Drag your program folder into the appropriate lab section folder. When you do this, the Mac will warn you that you will not be able to look at this folder. That is fine. Just click "OK."
- Log off of the computer before you leave.

You can submit your work up to 11 p.m. two days after your lab (11 p.m. Wednesday for those in the Monday Labs, and 11 p.m. Thursday for those in the Tuesday Lab). If you submit and later discover that your submission was flawed, you can submit again. We will grade the latest submission made before the 11 p.m. deadline. The Mac will not let you submit again unless you change the name of your folder slightly. It does this to prevent another student from accidentally overwriting one of your submissions. Just add something to the folder name (like "v2") and the resubmission will work fine.

Grading Guidelines

As on labs, we will evaluate your program for both style and correctness. Here are some specific items to keep in mind and focus on while writing your program:

Style
Descriptive comments
Good names
Good use of constants
Appropriate formatting

Design
Good use of boolean expressions, loops, conditionals
General correctness/design/efficiency issues
Parameters, variables, scoping
Proper use of arrays

Correctness
Moving
Turning
Eating
Growing
Dying

Summary of Nibbles Class

The following should give you a basic picture of how the `Snake` and `NibblesField` classes are used. Some details of the code have been omitted, but you can find them in the starter code on the web site.

You will not need to edit this class.

```
/**
 * Plays the game of Nibbles.  It draws a snake on the screen
 * and a piece of food.  The user uses the arrow keys to move
 * the snake and try to eat the food.  The snake dies if it
 * runs into the boundary or itself.
 */
public class Nibbles extends WindowController implements KeyListener {

    // The snake that moves around.
    private Snake snake;

    public void begin() {
        // build the field
        NibbleField field = new NibbleField(canvas);

        // create a snake with its head at a random location that is guaranteed
        // to allow the entire snake to fit in the field.
        snake = new Snake(field);

        ...
    }

    // Handle the arrow keys by telling the snake to go in the direction of the arrow.
    public void keyPressed(KeyEvent e) {
         ...
         if (e.getKeyCode() == KeyEvent.VK_UP) {
             snake.setDirection(Direction.UP);
         } else if (e.getKeyCode() == KeyEvent.VK_DOWN) {
             snake.setDirection(Direction.DOWN);
         } else if (e.getKeyCode() == KeyEvent.VK_LEFT) {
             snake.setDirection(Direction.LEFT);
         } else if (e.getKeyCode() == KeyEvent.VK_RIGHT) {
             snake.setDirection(Direction.RIGHT);
         }
         ...
    }
}
```

Snake Starter for Part I

```
public class Snake extends ActiveObject {

  // constants to control keyboard movement
  // the snake's growth rate
  private static final int AMOUNT_TO_GROW = 4;

  // initial length of snake
  private static final int INIT_LENGTH = 3;

  // pause time between movements
  private static final int PAUSE_TIME = 250;

  public Snake(NibbleField theField) {   /* WRITE THIS */  }

  public void setDirection(Direction dir) {  /* WRITE THIS */ }

}
```

Summary of Direction and Position for Part I

You won't need to edit these classes.

```
public class Direction {

    // Constant names used to describe directions outside this class
    public static final Direction UP
    public static final Direction DOWN
    public static final Direction LEFT
    public static final Direction RIGHT

    // Return the horizontal/vertical component of a direction
    public int getXChange()
    public int getYChange()

    // Check if one direction is the opposite of another
    public boolean isOpposite(Direction newDirection)

    // Return a string representation of a direction so that it can be output
    public String toString()
}

public class Position {
    // Create a new position.
    public Position(int theCol, int theRow)

    // Return the row or column part of the pair
    public int getCol()
    public int getRow()

    // Determine if two Positions are equivalent
    public boolean equals(Position pos)

    // Determine the position reached if you move in the specified Direction
    public Position translate(Direction dir)

    // Return a string representation of a position so that it can be output
    public String toString()
}
```

NibbleField Starter for Part II

This class is quite large, but you are only responsible for implementing the last four methods on page 13. While you need not study all of the details of this whole class, you may find looking at the rest of the code useful while designing those methods.

```
public class NibbleFieldPart2 {

  // Size of a single cell in the field
  private static final int CELL_SIZE = 14;

  // Minimum amount of blank space between field and canvas
  private static final int FIELD_INSET = 10;

  // Colors to use when drawing food or pieces of the snake
  private static final Color FOOD_COLOR = Color.red;
  private static final Color SNAKE_COLOR = Color.green;

  // Dimensions of the field measured in cells
  private int fieldWidth;
  private int fieldHeight;

  // The grid contains rectangles.  Null indicates an empty slot.  If a rectangle
  // is present, its color will indicate to the snake what it is.
  private FilledRect[][] field;

  // upper left corner of the field on the display
  private int cornerX, cornerY;

  // the drawing canvas
  private DrawingCanvas canvas;

  // Current location of the food
  private Position foodPos;

  // Random number generators used to place food.
  RandomIntGenerator rowGenerator, colGenerator;

  /**
   * Constructor
   */
  public NibbleFieldPart2(DrawingCanvas theCanvas) {
    canvas = theCanvas;

    cornerX = FIELD_INSET;
    cornerY = FIELD_INSET;

    fieldWidth = (int)(canvas.getWidth() - 2 * FIELD_INSET)/CELL_SIZE;
    fieldHeight = (int)(canvas.getHeight() - 2 * FIELD_INSET)/CELL_SIZE;

    // Create the array use to record contents of the field.
    field = new FilledRect[fieldWidth][fieldHeight];

    //draws the borders of the field
    new FramedRect(cornerX - 1, cornerY - 1, CELL_SIZE*fieldWidth + 1, CELL_SIZE*fieldHeight + 1, ca

    // Create random number generators used to position food
    rowGenerator = new RandomIntGenerator(0, fieldHeight - 1);
    colGenerator = new RandomIntGenerator(0, fieldWidth - 1);
```

```
  //draws the first initial food to "eat"
  placeFood();

}

/**
 * return true if the specified position in the field holds food
 */
public boolean containsFood(Position pos) {
  return pos.equals(foodPos);
}

/**
 * return true if the specified position in the field contains part of the snake
 */
public boolean containsSnake(Position pos) {
  return isOccupied(pos) && !containsFood(pos);
}

/**
 * return the position of the center of the field
 */
public Position getCenter() {
  return new Position(fieldWidth/2, fieldHeight/2);
}

/**
 *  Move the food to a new location on the field
 */
public void consumeFood() {
  Position oldPos = foodPos;
  placeFood();
  removeContents(oldPos);
}

// Display a piece of the snake as a rectangle at a specified position
// Parameters:
//    pos - row and column of the cell to contain the rectangle
public void addItem (Position pos) {
  if (!outOfBounds(pos)) {
    if (isOccupied(pos)) {
      System.out.println("WARNING:  Adding item to " + pos + " but it is not empty!");
      Thread.currentThread().dumpStack();
    }
    showContents(pos, SNAKE_COLOR);
  }
}

// remove a piece of the snake from the field.  Complains if the position is already
// empty.
// Parameters:
//    pos - row and column of the cell to remove the item from
public void removeItem (Position pos) {
  if (!isOccupied(pos)) {
    System.out.println("WARNING:  Removing item from " + pos + " but it is empty!");
    Thread.currentThread().dumpStack();
  }
  removeContents(pos);
}
```

```
  /**
   * Place a new piece of food on the screen somewhere away from snake
   */
  private void placeFood() {
    // Randomly generate locations until we find an empty one.

    foodPos = new Position(colGenerator.nextValue(), rowGenerator.nextValue());
    while (isOccupied(foodPos)) {
      foodPos = new Position(colGenerator.nextValue(), rowGenerator.nextValue());
    }
    showContents(foodPos, FOOD_COLOR);

  }


  /************ YOU MUST WRITE THE LAST FOUR METHODS *********/

  /**
   * returns whether a position is out of the bounds of the field
   */
  public boolean outOfBounds(Position pos) {
    return false;  // CHANGE
  }

  /**
   * returns whether a particular field position is occupied
   */
  private boolean isOccupied(Position pos) {
    return false;  // CHANGE
  }

  /**
   * Remove the rectangle representing the food or a piece of the snake
   */
  private void removeContents(Position pos) {
    // FILL THIS IN
  }

  // Add a rectangle representing the food or a piece of the snake
  // Parameters:
  //    pos - row and column of the cell in which to place the item
  private void showContents(Position pos, Color hue) {
    // FILL THIS IN
  }

}
```