CS 51 Homework Laboratory # 10
Nibbles

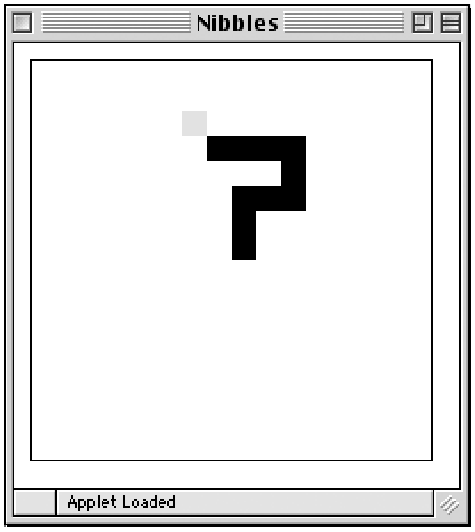
Objective: To gain more experience with lists and with method design.

—

The Problem This assignment asks you to implement the game of Nibbles. Nibbles is a snake. It moves around a field, looking for food. Unfortunately, it's 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.

As usual, start by dragging a copy of the starting folder from the server.

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 below and to the right of the food. It was moving to the left on 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.



If you want to see how to win at nibbles, go to http://9gag.com/gag/aOq7X16.

Design The Nibbles program consists of 5 classes/objects:

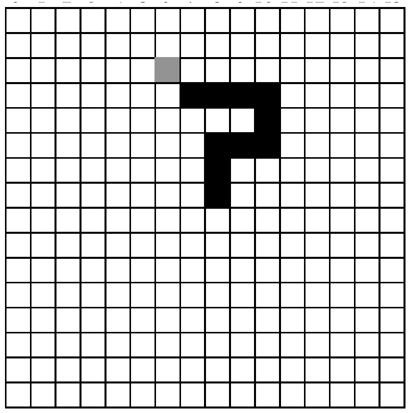
- `snake` - a class that creates and then animates the snake as it slithers around the screen,
- `nibbleField` - a class used to represent the 2D grid that the snake lives within,
- `nibblesGame` - the main program that initializes the nibble field and snake and handles the user input,
- `position` - 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.

We will give you working code for all but the `snake` and `nibbleField` class. The `nibbleField` class is mainly completed, but we will ask you to fill in the following methods that have not been completed: `showContentsAt`, `isOccupied`, `outOfBounds`, and `removeContents`. To help you test these methods, we have provided you with a program `nibbleFieldTester` that can be used to test whether these methods work properly.

Later we will ask you to implement the snake using our implementations of the other four classes.

The `nibblesGame` object: The `nibblesGame` object begins by creating the field and the snake, and then `start`ing the snake. It then receives the clicks of the arrow buttons and tells the snake to change direction accordingly. You will not need to touch this class.

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 grid 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 3 and column 7, and the head of the snake is found at row 4 and column 8. We can write these positions as (3,7) and (4,8). The `position` class is used to combine such a pair of grid coordinates into a single object much as an object of the `Point` 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 (the *row*) before the number that describes its horizontal position (the *column*). This is the opposite of the convention used with canvas coordinates.

The `position` constructor, which generates an object of type `FieldPosition`, takes a row and a column as parameters:

```
positionRow (r) col (c)
```

In addition to this constructor, the `FieldPosition` type (and hence the `position` class) provides the following methods:

```
row -> Number
col -> Number
== (other: FieldPosition) -> Boolean
// return a position one cell in dir direction from self
translate (dir: Direction) -> FieldPosition
asString -> String
```

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

Note that we do not usually list `asString` in the type for an object as it is automatically included in every type. The default method provided is inherited automatically from a class `graceObject`, which is a superclass of all classes and objects. The default `asString` does not provide much of interest so we try to override it whenever we might need it.

In earlier programs we sometimes defined `==` using the built-in confidential method `isMe`, which just checks to see if the argument is the same object. E.g., we often write something like:

```
method == (other: SomeType) -> Boolean {
   isMe (other)
}
```

However, in some cases, that is not what we want.

The Position class is an example of where we want something different. We want to say two positions are equal if they have the same row and column. This is an example of what is sometimes called extensional equality, checking to see if two objects are the same by comparing their externally visible properties are the same. The code to check that is given below:

```
// check to see if contents of two positions are the same
method == (other: FieldPosition) -> Boolean {
  (row == other.row) && (col == other.col)
}
```

The `direction` Class: The class `direction` (generating objects of type `Direction`) is used to encode the direction in which the snake is currently moving. There are four possibilities: `up`, `down`, `left` and `right`.

Internally, the representation of a `direction` object is similar to that used for `Points`. A `Direction` represents 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 `FieldPosition` type rather than with the `Direction` type.

```
  translate (dir: Direction) -> FieldPosition
```

Given a `FieldPosition` 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:

```
   rowChange -> Number
   colChange -> Number
   isOpposite (newDir: Direction) -> Boolean
```

The `rowChange` and `colChange` 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: An object of type `NibbleField` (generated by class `nibbleField`) represents the actual contents of the game grid. Most of the necessary information is encoded using a `Matrix` containing one entry for every cell in the grid. The entries of the grid can hold `filledRect`s corresponding to the snake or food. If neither the snake or the food occupies the slot, the that slot of the grid is filled with the object `empty`, which has type `FieldObject`, which just has methods `removeFromCanvas`, `asString`, and `==`. Note that filled rectangles also have this type, as they support each of those operations. Hence the field will be represented as a `Matrix` holding elements of type `FieldObject`, and it will initially be filled with default value `empty`. If a slot has a value different from `empty` during the play of the game, it will refer to the filled rectangle 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.

```
   class nibbleFieldOn (canvas: DrawingCanvas) -> NibbleField
```

The initialization code uses the `width` and `height` 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 the following methods:

```
type NibbleField = {
  // eat food and put food in new (unoccupied) place on the field
  consumeFood -> Done

  // Indicate whether something (snake or food) is at that posn of field
  isOccupied(posn: FieldPosition) -> Boolean

  // Does posn contain food?
  containsFood (posn: FieldPosition) -> Boolean

  // Does posn contain the snake?
  containsSnake (posn: FieldPosition) -> Boolean

  // Is posn a legal position for the snake on the field (i.e., within boundaries)
  outOfBounds (posn: FieldPosition) -> Boolean

  // Return position in center of the field
  centerPosition -> FieldPosition

  // Add an item to posn in field.  Must be unoccupied or will get an error!
  addItem (posn: FieldPosition ) -> Done

  // Remove an item from posn.  If nothing there will get an error.
  removeItem (posn: FieldPosition) -> Done
}
```

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 should 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 or coordinated 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, as once the snake has moved into that cell the `nibbleField` will no longer contain any trace of 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 `centerPosition`.

The `snake` class: The most interesting class is the `snake`. The `snake` class, which generates an object of type `Snake`, will be animated, of course. 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.

```
type Snake = {
  // start it moving
  start -> Done

  // change direction of snake movement to dir
  direction:= (dir: Direction) -> Done
}
```

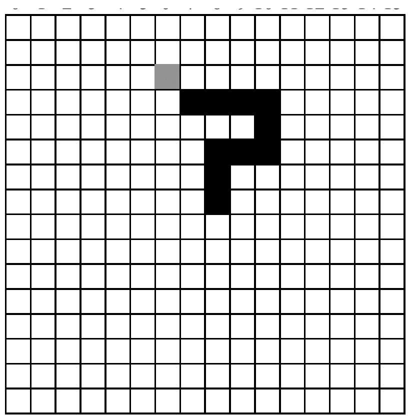
If the user presses an arrow key, the `nibbleGame` object gets notified of the event, and the code associated with the invocation of `onKeyPressDo` calls the `direction:=` method of the snake to inform the snake that the direction in which it moves should be changed. The snake should ignore the controller if the new direction is the opposite of the snake's current direction.

The snake moves and tries to eat as part of the code within its `start` 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 `FieldPosition`s of the parts of the snake's body in a list.

Each element of the list in the `Snake` class identifies the `FieldPosition` 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 list of `FieldPosition`s would be used to describe it. The picture below suggests what the list would look like. As a shorthand in this figure, we have written pairs like (4,8) and (6,11) to represent `FieldPosition` values, but you should realize that you can't actually do this in your code. Instead of (3,7) you would have to say `positionRow (4) col (8)`.

Table 1: default

(4,8)	(4,9)	(4,10)	(5,10)	(6,10)	...	...	...	...	...

The cell at one end of the snake is in row 4, column 8 and this `FieldPosition` is stored in the first element of the list. The piece next to this piece is at row 4, column 9 and its `FieldPosition` is shown in the second element of the list. The positions of the remaining pieces of the snake's body are placed in the remaining cells of the list in consecutive order.

You can't tell from either the picture of the game grid or the `FieldPosition` list 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 list in either order as long as you remember which order you decided to use while writing the code for your `snake` class.

Depending on which order you hold the snake you will use either `addFirst` and `removeLast` or `addLast` and `removeFirst` list operations.

Be sure to examine the cell where the snake is moving before you move. *Once it has moved it will be impossible to determine whether it contained the snake or food.*

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. Similarly, 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.

Think about the design of the `start` method carefully in your design. A well thought-out animated while loop can make the program relatively simple, while if you just throw it together you will get a very badly designed mish-mash of confusing (and incorrect) code.

Your Job This lab has two parts. For the first part, you will fill in the bodies of the four missing methods of class `nibblesField`: `showContentsAt`, `isOccupied`, `outOfBounds`, and `removeContents`.

The first of these is confidential and is called by method `addItem`, whose code is included. The header is

```
method showContentsAt (posn: FieldPosition) toBe (contentColor: Color)
                                      is confidential
```

It should create a filled rectangle with color `contentColor` at `posn` on the field.

The next two methods return a boolean to indicate whether the given position is occupied by a colored rectangle or if it is not a legal position on the board. (The number of rows and columns on the board is given by `numCols` and `numRows`, whose values are computed in the class.) The last removes an item from the board and replaces it with an empty field object.

As noted earlier, the program `nibbleFieldTester` should be used to determine if your new methods work correctly. When `nibbleField` is working correctly the program will pop up a window with a piece of food and two green cells. Code that starts out commented out can be uncovered and used to test the other methods you will write.

For the second part of the lab you are to write a complete version of the `snake` class. Be sure to load the files `Matrix.grace`, `NibblesGame.grace`, `Snake.grace`, and `Position.grace` into the Grace editor. However, you should only make changes to `Snake.grace`.

The snake constructor is passed the `NibblesField` that it wanders through as its only parameter. The constructor should create and initialize the list to hold the positions of its body. It should use the field's `centerPosition` 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 `start` 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.

The things you need to do before coming to the lab are described in the Lab Design Form. There are a lot more things to worry about that are not on that document (e.g., adding the missing methods in `nibbleField`), but that will help you focus on some of the important parts of the problem. If you would like feedback earlier, we encourage you to discuss your design with the instructor any time before the lab. Be sure to carefully read the code in our incomplete `nibbleField` class before coming to lab. Our incomplete `nibbleField` class is available from the link here:
http://www.cs.pomona.edu/classes/cs051G/labs/nibbles/NibbleField.grace.

The starter code for the `snake` class is available at:
http://www.cs.pomona.edu/classes/cs051G/labs/nibbles/Snake.grace.

For this lab, you are allowed to work with a partner on the design prior to lab. You can discuss the problem with your partner and sketch out some ideas, but you should not write code together. Each of you should have your own copy of the design, but put both authors' names on it.

Here are some other things to think about that are not included directly on the design form, but will be necessary for you to successfully complete the program. We suggest that you write this information out before the lab so you do not have to waste lab time on them.

- headers for the methods you will define during the lab. The header should describe the purpose of the method, the parameters it takes, the meaning of its return value if it has one, and a brief English description of what the method will do.
- descriptions of all the methods you will define, taking special care with how the start, stretch and shrink methods of `Snake` will operate. Be sure to describe the looping and if statements you will use to structure your code as well as how you will manipulate your list in these methods.
- declarations for constants, instance variables, and parameters

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 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 `start` 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 animated while loop of the start method.

Once that is working, use the values passed in the `direction:=` 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 likely 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.

READ THIS NOTE: If you try instead to go into a quick loop and stretch three times you code WILL NOT WORK! It will neglect to test all of the possibilities where the snake can run into contingencies – including biting itself, hitting a wall, and eating more food. In nearly fifteen years of assigning this problem, I have not seen a single student succeed who took this path. Maybe you will be that student! Maybe someone will give you a million dollars tomorrow! However, neither is likely. If you ignore this advice you will waste many hours before you finally realize your solution will NOT WORK. If you can afford the time, be my guest. If not, consider my suggestion of skipping shrink steps rather than introducing another loop.

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.

Another warning: If you try to examine a cell at a row and column that don't exist in your matrix, the program will crash. Thus you should always check if a `FieldPosition` is `outOfBounds` before asking whether that position contain the snake or the food. If you use an "and" statement to check two things, be sure to check for out of bounds first, and use the form:

```
firstCheck && {secondCheck}
```

rather than `firstCheck && secondCheck`. The first form (with the extra curly braces to the right of the `&&`) will only evaluate `secondCheck` if it first determines `firstCheck` is true. This can help keep your program from crashing if evaluating the firstCheck can prevent `secondCheck` from crashing.

As an example of what can go wrong, imagine evaluating `(i > 0) && ((l.at (i) == 16)`. Now suppose that the value of `i` is -1. Evaluating `i > 0` will result in `false`, but when the second half of the expression is evaluated, it will result in trying to access `l.at (-1)`, which doesn't exist – causing the system to crash.

If instead you were to write `(i > 0) && {l.at (i) == 16}`, it will evaluate `i > 0`, determine that it is false, and then not bother to evaluate the second part. This is sometimes called short-circuit evaluation. If the first part of an `&&` expression is false, there is no need to evaluate the expression as it can't possibly make the entire expression true. Thus the short-circuit version of the operation (the one using curly braces on the right-hand expression) will just return false without evaluating the second part.[1] You will need to use the short-circuit version of `&&` in this program so that you can first check to see if a position is out of bounds before looking at what is actually there.

Submitting Your Work The lab is due Monday at 11 p.m. as usual. When your work is complete you should deposit it in the appropriate dropbox. Give it a name of the form `lab10_lastnamefirstname`. Also make sure that your name is included in the comment at the top of all the files you turn in.

[1]There is a similar short-circuit version of `||`. The expression `p || {q}` returns true without evaluating `q` if `p` is true.

Table 2: Grading Guidelines

Value	Feature
	Design preparation (4 pts total)
1 pt.	stretching & shrinking
2 pt.	start method of snake
1 pt.	testing
	Readability (5 pts total)
2 pts.	Descriptive comments
1 pts.	Good names
1 pts.	Good use of constants
1 pt.	Appropriate formatting
	Code Quality (6 pts total)
1 pt.	conditionals and loops
2 pt.	General correctness/design/efficiency issues
1 pts.	Parameters, variables, and scoping
2 pts.	Good correct use of lists
	Correctness (5 pts total)
1 pt.	Moving
1 pt.	Turning
1 pt.	Eating
1 pt.	Growing
1 pt.	Dying