Assignment 2

ECSE 250 Fall 2023

posted: Oct. 30, 2023
due: Nov. 14th, 2023

General Instructions

For this assignment need to download the files provided. Your task is to complete and submit the following files:

```
MySinglyLinkedList.java
MyStack.java
Caterpillar.java
```

- **Do not change any of the starter code that is given to you. Add code only where instructed, namely in the "ADD YOUR CODE HERE" block.** You may add `private` helper methods to the class you have to submit, but you are not allowed to modify any other class.
- Please make sure that the file you submit is part of a package called `assignment2`.
- You are NOT allowed to use any class other than those that have already been imported for you. **Any failure to comply with these rules will give you an automatic 0.**
- Do NOT start testing your code only after you are done writing the entire assignment. It will be extremely hard to debug your program otherwise. If you need help debugging, feel free to reach out to the teaching staff. When doing so, make sure to mention what is the bug you are trying to fix, what have you tried to do to fix it, and where have you isolated the error to be.

Submission instructions

- Submissions will be accepted up to 3 days late. Note that submitting one minute late is the same as submitting 3 days late. We will deduct points for any student who has to resubmit after the due date (i.e. late) irrespective of the reason, be it the wrong file submitted, the wrong file format submitted or any other reason. We will not accept any submission after the 3 days grace period. This policy will hold regardless of whether or not the student can provide proof that the assignment was indeed "done" on time.
- Don't worry if you realize that you made a mistake after you submitted: you can submit multiple times but **only the latest submission will be evaluated**. We encourage you to submit a first version a few days before the deadline (computer crashes do happen and Ed Lessons may be overloaded during rush hours).

- **Do not submit any other files, especially .class files and the tester files. Any failure to comply with these rules will give you an automatic 0.**
- Whenever you submit your files to Ed, **you will see the results of some exposed tests**. If you do not see the results, your assignment is not submitted correctly. **If your assignment is not submitted correctly, you will get an automatic 0. If your submission does not compile on ED, you will get an automatic 0.**
- The assignment shall be graded automatically on ED. **Requests to evaluate the assignment manually shall not be entertained, and it might result in your final marks being lower than the results from the auto-tests.** Please make sure that you follow the instructions closely or your code may fail to pass the automatic tests.
- The exposed tests on ED are a mini version of the tests we will be using to grade your work. If your code fails those tests, it means that there is a mistake somewhere. Even if your code passes those tests, it may still contain some errors. Please note that these tests are only a subset of what we will be running on your submissions, we will test your code on a more challenging set of examples. Passing the exposed tests assures you that your submission will not receive a grade lower than 40/100. We highly encourage you to test your code thoroughly before submitting your final version.
- Next week, a mini-tester will also be posted. The mini-tester contains tests that are equivalent to those exposed on Ed. We encourage you to modify and expand it. ***You are welcome to share your tester code with other students on Ed.*** Try to identify tricky cases. Do **not** hand in your tester code.
- **Failure to comply with any of these rules will be penalized**. If anything is unclear, it is up to you to clarify it by asking either directly a TA during office hours, or on the discussion board on Ed.

Learning Objectives

In this assignment, you will get some experience working with linked lists and stacks. You will implement a data structure for a fun application. Starting with this assignment we will start to focus also on the efficiency of your algorithms. You will learn to look at code with a more critical eye, without only focusing on the correctness of your methods.

PART I: Data Structure

In the first part of the assignment, you will implement a SinglyLinkedList and a Stack, as seen in the class. Note that, if you have implemented the practice problems in the tutorials, your Part I is almost done.

Instructions and Starter Code

In package `assignment2`, a `MySinglyLinkedList.java` is provided with some starting code. **Do not modify anything that was provided; otherwise, your program will fail the testers**. This class implements a singly linked-list with a generic type. The `MySinglyLinkedList<E>` is a class with the following content.

- The class already has a `private` class `SNode`. The `SNode` class contains two private fields
  - An `E` named `element` indicating the element of the node
  - A `SNode` named `next` indicating the next node
- The class already has the following `public` fields:
  - An `int` named `length` indicating the number of nodes in the list
  - A `SNode` named `head` indicating the head of the list
  - A `SNode` named `tail` indicating the tail of the list

  Note that, the fields have been left `public` just to make testing easier on our side. As discussed in class, in general, you would want to keep as much as you can private and instead provide `public` getters/setters when needed.
- The class already has a private class `SLLIterator` that implements the `Iterator<E>` interface (from `java.util.Iterator`) which allows the user to iterate over the list.
- The class already has an `iterator` method that returns a `SLLIterator` object.

In package `assignment2`, you will also find a `MyStack.java`, which is provided with some starting code. The `MyStack<E>` class has a `private` fields, a `MySinglyLinkedList<E>`, to store the elements of the stack. You will also find the `toString` already implemented for you.

Methods you need to implement

The first task is to complete the `MySinglyLinkedList<E>` class. The `MySinglyLinkedList<E>` class must also have the following `public` methods:

- A constructor that initializes `size` of the list to 0 and set `head` and `tail` to null.
- A `addFirst` and a `addLast` method which takes input as an object of generic type and adds the element as the first/last element of the list. The method returns `true` if the operation is successful or `false` otherwise.

- A `removeFirst` and a `removeLast` method which removes the first/last node from the list and returns the element. The method should throw a `NoSuchElementException` if the list is empty.
- A `peekFirst` and a `peekLast` method which returns the first/last element without removing it from the list. The method should throw a `NoSuchElementException` if the list is empty.
- A `clear` method that removes all elements on the list.
- A `isEmpty` method that returns true if the list is empty and false otherwise.
- A `getSize` method that returns the size of the linked list.

Lastly, you can implement extra helper methods for your algorithms. You can also override the `toString` method for you to visualize your list.

Complete the `MyStack<E>` class. This class must also have the following `public` methods:

- A constructor that takes no inputs and creates an empty stack.
- A `push` method that takes an input of type `E` and adds the input to the top of the stack. The method returns `true` if the operation is successful or `false` otherwise.
- A `pop` method that removes the top of the stack and returns the element. The method should throw a `NoSuchElementException` if the stack is empty.
- A `peek` method that returns the element at the top of the stack without removing it. The method should throw a `NoSuchElementException` if the stack is empty.
- A `isEmpty` method that returns true if the stack is empty and false otherwise.
- A `clear` method that clears the stack (i.e. removes all the elements from the stack).
- A `getSize` method that returns the number of elements in the stack.

PART II: The Very Hungry Caterpillar

Introduction[1]

Figure 1: Gus. An AI generated image from Canva's Magic Media tool, created from the prompt 'Gus, a colorful caterpillar with a big appetite and an adventurous spirit'.

Once upon a time, in a lush colorful meadow nestled between the rolling hills of Munchgill Valley, there lived a peculiar caterpillar named Gus. Now, Gus was no ordinary caterpillar. He had an insatiable appetite and an adventurous spirit that was as boundless as his stomach was bottomless.

Gus's favorite pastime was munching on delectable delicacies, and he had a particular fondness for trying new foods. One fine day, as he slithered along the meadow, he stumbled upon a magical patch of food that seemed to have an endless variety of treats - from succulent strawberries to Swiss cheese, and even shimmering stardust-infused snacks!

This enchanted patch of food, however, came with a twist. Each time Gus indulged in a new treat, he magically transformed, growing longer, shrinking, or flipping depending on the bite. And so, our gluttonous Gus embarked on a gastronomic journey to devour his way to greatness, one tasty morsel at a time.

Unbeknownst to Gus, a group of brilliant young minds from the Munchgill Valley University of Magic had been observing his peculiar behavior. These aspiring wizards and computer scientists were tasked with a peculiar challenge by the wise old sage, Professor: to create a game that mirrored Gus's whimsical escapades, inspired by the beloved tale of "The Very Hungry Caterpillar".

In this magical realm, where learning was as enchanting as the spells cast by the wizards, the students were introduced to the concept of singly linked lists. The challenge was to represent Gus's ever-changing body using a singly linked list of segments, each segment symbolizing a delightful bite of food that Gus devoured.

The students were thrilled by the opportunity to weave their coding spells and bring Gus's hilarious, food-filled adventures to life. They set out on their quest, armed with the knowledge of singly linked lists, ready to face the challenges and twists that Gus's gluttonous journey had to offer.

And so, with their wands (or rather, keyboards) at the ready, the students delved into the whimsical world of Gus the Gluttonous Caterpillar. Little did they know that this coding adventure would not only test their skills but also lead to a hilariously entertaining game, filled with laughter, learning, and the occasional uncontrollable giggle as Gus grew longer with every delicious bite. The fate of Gus's gluttonous escapades now rested in the hands of these budding wizards and computer scientist, ready to craft a game that would leave a lasting mark on the magical meadow of Munchgill Valley.

[1] This background story was created with the help of https://chat.openai.com

Instructions and Starter Code

As mentioned in the section before we will use a singly linked list to represent a caterpillar. The starter code contains many files, divided into different packages. The `assignment2` package contains the following classes:

- A `Position` class, which is defined by the x and y coordinates. Throughout this pdf, we will represent such objects using the ordered pair `(x, y)`.
- A `GameColors` class that defines some possible colors of a segment.
- A `Segment` class, which contains the following:
  - A `Position`, representing where on the board this segment is.
  - A `Color`, representing the color of this specific segment. The colors of the segments will mostly depend on what the caterpillar has eaten in order to have gained those segments.
  - A `toString` method is overridden so that you can display a segment.
- An `EvolutionStage` that contains an enumeration of the evolution stage in which the caterpillar finds itself. An enumeration in java is a special data type that represents a fixed set of constants. You can check out EvolutionStage.java to see how an `enum` is defined. All you need to know for this assignment is that you can compare enumeration values using `==` and `!=`. For instance, `stage == EvolutionStage.FEEDING_STAGE` is a valid boolean expression.
- `assignment2.food` – This package contains all the classes representing the possible food items the caterpillar might encounter while slithering along the meadow. You can look inside these files to see what can be useful to you, but you should not modify them!

A `Caterpillar` class that represents the caterpillar as a singly linked-list of `Segment`s. This is done by extending `MySinglyLinkedList` class with type `<Segment>`. You need to modify this file in order to complete this part. You will need to understand and use the other classes contained in this package, but you should not modify them! Aside from the field inherited, each caterpillar is defined by the following fields:

- `EvolutionStage stage` : the stage in which the caterpillar finds itself.
- `MyStack<Position> positionsPreviouslyOccupied` : all the positions previously occupied by the caterpillar with the most recent one on the top of the stack (this will become more clear once you read the description of the methods below). We will represent the stack as a list of elements between square bracket: the element at the top of the stack will be the right most one, while the element at the bottom of the stack will be the left most.
- `int goal` : the number of segments the caterpillar aspires to in order to be able to become a wonderful butterfly.
- `int turnsNeededToDigest` : the number indicating how many turns are left before the caterpillar is ready to take its next bite (see the method definitions below for more information).

Finally, inside the `Caterpillar` class you can also find a static field called `randNumGenerator` of type `Random`. To use it, you must follow the following guidelines:

- Do **NOT** assign a new reference to this variable. In fact, you should never in your code create a new `Random` object.
- If asked to generate a random number you must use `randNumGenerator`.
- You should generate random numbers only when it is strictly necessary.

If in your code you fail to respect the guidelines above, your methods will probably not behave how we expect them to, leading to some tests possibly failing.

Please note that in the `Caterpillar` class there's a `toString()` method that has been implemented to make debugging a little easier for you. The method returns a string representing a caterpillar. The string contains the positions of each of the segments. The right most pair of coordinates represents the position of the head, while the left most represents the position of the tail. The color of the segment is used to color the text. So, for example, if we have a caterpillar with a red head in position `(3,1)`, a second yellow segment in position `(2,1)`, and its last blue segment in position `(2,2)`, then `toString()` will return the following string:

(2,2) (2,1) (3,1)

Tips and Safe Assumptions

Before starting to work on the implementation of the methods that were left to you, please keep the following in mind:

- Read through the description of all of the methods before starting to code. This will give you an idea of the general assignment and might help you decide on how to better organize your code (e.g. what could be a good `private` helper method to add?)
- You will see that the method called `eat()` has been overloaded. I have listed each of the method's variants in order of difficulty. You don't need to implement them in that order, but this is what I would suggest you to do.
- You will see that all the fields of the `Caterpillar` class have been left `public` just to make testing easier on our side. As discussed in class, in general, you would want to keep as much as you can private and instead provide `public` getters/setters when needed.
- As already mentioned before, make sure to test and debug your methods while you are writing it. **If you wait to debug your code once you are done writing the entire assignments it will be very hard to do so effectively**. Please note that you should be able to test your methods on your own. You should not wait for or solely rely on the exposed tests we provide.
- Throughout the assignment you can always assume the following:
  - The caterpillar will always *move* before attempting to *eat*.
  - The caterpillar will *eat* only when it is in a feeding stage.
  - The methods will be tested only on valid inputs: `null` is not a valid input for any of the methods!

Methods you need to implement

To complete your task you need to implement all of the methods listed below. See the starter code for the full method signatures. Your implementations must be efficient. For each method below, we indicate the worst-case run time using $O()$ notation.

- `Caterpillar(Position p, Color c, int goal)` : creates a caterpillar with one single segment of the given color, in the given position. The input `goal` is used to set the number of segments the caterpillar needs to acquire before becoming a butterfly. Whenever a caterpillar is first created, it finds itself in a feeding stage, ready to devour every delicacy within its reach.

  This method runs in $O(1)$.

- `getSegmentColor(Position p)` : returns the color of the segment in the given position. If the caterpillar does not have a segment placed upon the given position, then the method returns `null`.

  This method runs in $O(n)$, where n is the number of segments.

- `eat(Fruit f)` : this foundational food group is the basis of growth for our wiggly friend. With each fruit bite, the caterpillar grows longer by getting a new segment added matching the color of the fruit ingested. The new segment should be added at the tail of the caterpillar, and its position should be the most recent position previously occupied by the caterpillar. Make sure to update all relevant fields to represent this growth.

  This method runs in $O(1)$.

- `eat(Pickle p)` : the sourness of the pickle makes the caterpillar retrace its steps. This method updates its segments accordingly.

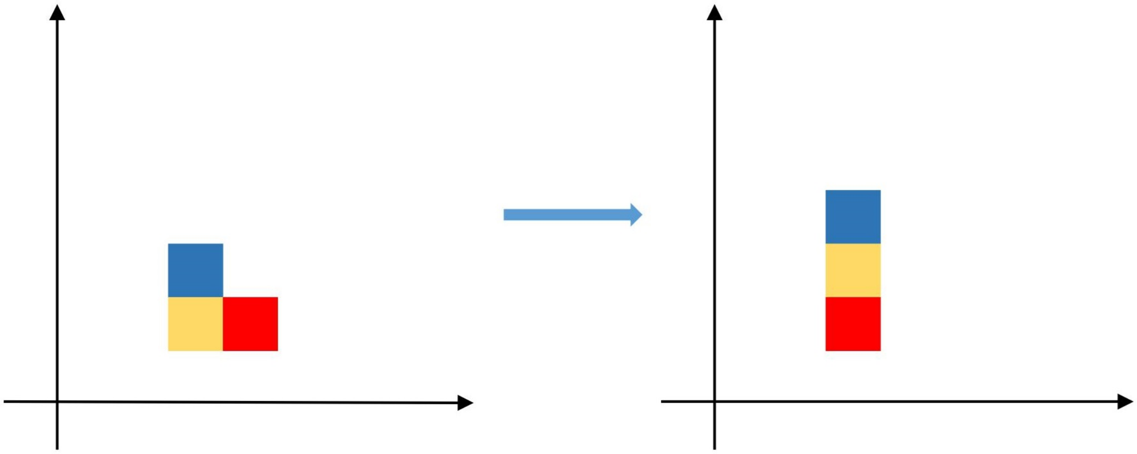
  For instance, suppose we have the following caterpillar:

  (2,2) (2,1) **(3,1)**

  And suppose that the most recent position previously occupied by this caterpillar is `(2,3)`. Then, after eating a pickle the caterpillar will have the following segments:

  (2,3) (2,2) **(2,1)**

  It might be easier to picture those segments in a Cartesian plane. If you do so, this is how the caterpillar would change in the example above:



This method runs in $O(n)$, where n is the number of segments.

- `eat(Lollipop lolly)` : this swirl of colors makes our caterpillar longing for a playful makeover. Shuffle all the caterpillar's colors!

  There are different ways of doing this, but for this assignment you will need to implement the method using the Fisher–Yates shuffle algorithm. The algorithm runs in $O(n)$ using $O(n)$ space, where n is the number of segments. To perform a shuffle of the colors follow the steps:

  – Copy all the colors inside an array

  – Shuffle the array using the following algorithm:

```
for i from n-1 to 1 do
    j <-- random integer such that 0 <= j <= i
    swap a[j] and a[i]
```

  To generate a random integer use the `Random` object stored in the class field called `randNumGenerator`.

  – Use the array to update the colors in the all of the segments.

- `eat(IceCream gelato)` : the caterpillar did not expect this deliciously looking, creamy treat to be sooo cold! Its body does a hilarious flip, reversing on itself and its (new) head turns blue like an icicle. At this point it lost track of where it has been before, and the stack of previously occupied positions is now empty.

  This method runs in $O(n)$, where n is the number of segments.

- `eat(SwissCheese cheese)` : who can resist the cheesy delight of those perfectly shaped eyes![2] But wait...by indulging on those cheesy holes the caterpillar body is getting "holey" too. This method shrinks the caterpillar who loses every other segment of the its body. This means that the segments left will have the colors of every other segment of the original body. But be careful, we do not want a caterpillar in pieces! The segments should still appear in positions

[2] Yes, holes in cheese like Swiss Cheese are actually called 'eyes'!

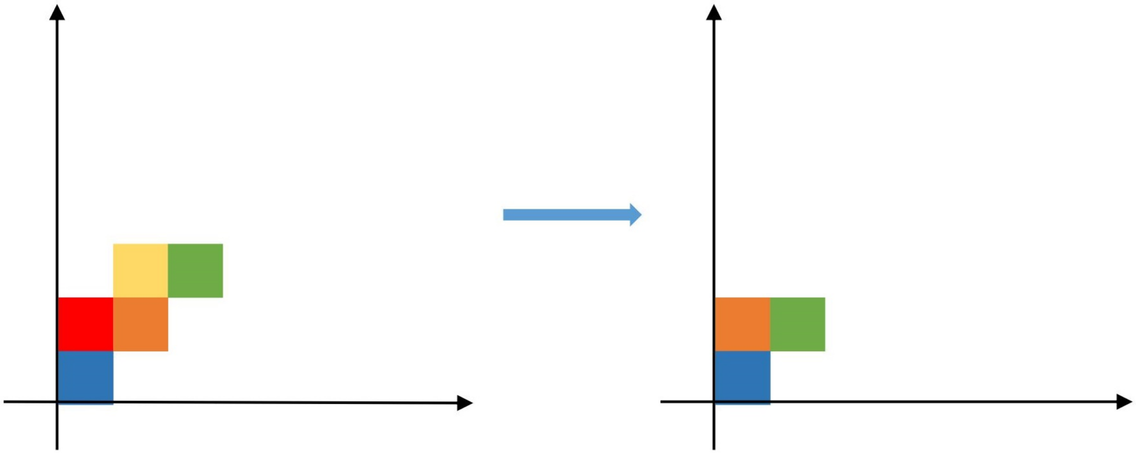
that are adjacent to one another, specifically the head will remain in the original position and only the first half (rounding up) of the segments' positions will be maintained. For example, consider the following caterpillar:

(2,0) (1,0) (1,1) (0,1) (0,0)

After eating Swiss cheese, the caterpillar will become as follows:

(1,1) (0,1) (0,0)

Or, using a Cartesian plane, this is how the change to the caterpillar can be represented:



Pay attention when updating the stack of positions previously occupied by the caterpillar. In the example above, if the stack was originally empty, after eating the cheese the stack will contain the positions `[(2,0), (1,0)]`, with `(1,0)` being at the top of the stack.

This method runs in $O(n)$, where n is the number of segments.

- `eat(Cake cake)` : the holy grail of all treats! Here's something that will make the caterpillar truly grow. When eating a cake, the caterpillar enters its `GROWING_STAGE`. Its body will grow by as many segments as the energy provided by the cake. These segments will have a random color and will be added at the tail of the caterpillar's body. Be careful though, this growth might not take place entirely in this method! In fact, the caterpillar will grow by a number of segments that is equivalent to the minimum between the energy provided and the number of previously occupied positions that are still available to the caterpillar. For example, consider the following caterpillar:

  (0,1) (0,0) (1,0)

  and assume the cake eaten provides an energy equal to `3` while the stack of previously occupied positions contains `[(2,0), (1,0), (1,1)]` (the right most element being at the top of the stack). From eating the cake, the caterpillar should grow by 3 segments, but it is not possible to do so in this method. Only `(1,1)` can be used for growth by the caterpillar, since `(1,0)` has already been occupied by another caterpillar's segment.

If the cake's energy cannot be all consumed by the caterpillar in this method, the "left over" is then stored in the field `turnsNeededToDigest`, as the caterpillar will need that many turns in order to fully digest the cake.

There are two more things you should keep in mind: it is possible for the caterpillar to reach its butterfly stage while growing in this method. If this is the case, its stage should be updated and the method should terminate right away. It is also possible for the caterpillar to fully consume the energy provided by the cake right here in this method. If this is the case, and the butterfly stage has not been reached, then the feeding stage should be resumed.

Note: to generate a random color generate a random index that you can then use to select a color from the array `GameColors.SEGMENT_COLORS`.

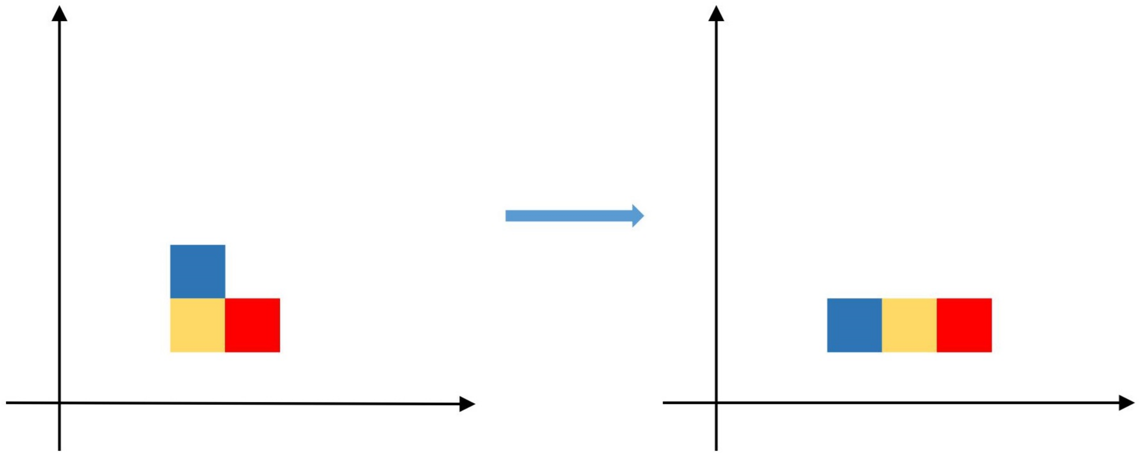
This method runs in $O(n*m)$, where n is the number of segments and m the energy provided by the cake.

- `move(Position p)` : if possible the caterpillar moves its head (and all its body) to the input position. If the input position is out of reach, i.e. if when seen as a point on a Cartesian plane it is not orthogonally connected to the head's position, then an `IllegalArgumentException` should be raised. If the position is within reach, but it is already occupied by a segment of the caterpillar's body, then moving will generate a collision leading to the caterpillar being in an `ENTANGLED` stage (unfortunately, caterpillars do not recover from this stage and the game will end). If on the other hand, the position is reachable and moving to it would not lead to a collision, then the body of the caterpillar should be updated accordingly: all the positions in the segments should be updated to represent the caterpillar moving forward to the input position, while the colors remain the same. For example, below is the representation of a caterpillar before and after the move to the position `(4,1)`:

  (2,2) (2,1) (3,1)

  (2,1) (3,1) (4,1)

  Or, using the representation with the Cartesian plane, this is how the caterpillar would change:



  The method should update the stack keeping track of the previously occupied positions accordingly (e.g., in the example above, the top of the stack would have become `(2,2)`).

Finally, if the caterpillar is still digesting a cake it had previously eaten, a segment with a random color should be added at the tail of its body ensuring its ever-growing journey continues. This in turn should decrease the number of turns left for its digestion, and might update its evolution stage to `BUTTERFLY` (if the goal was reached). If the caterpillar is in a growing stage, but its digestion is fully completed, then the `FEEDING_STAGE` should be resumed since caterpillar is now ready to venture out for more treats.

This method runs in $O(n)$, where n is the number of segments.

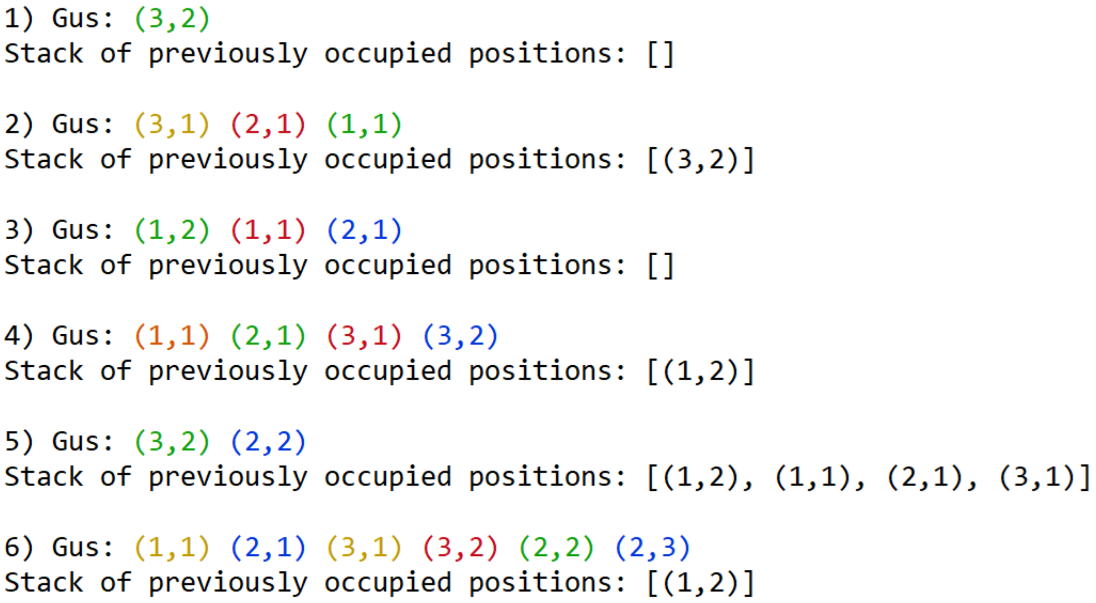
A small example

Here is a small example of how some of these methods might get called and execute. Consider the following snippet of code:

```
Position startingPoint = new Position(3, 2);
Caterpillar gus = new Caterpillar(startingPoint, GameColors.GREEN, 10);

System.out.println("1) Gus: " + gus);
System.out.println("Stack of previously occupied positions: " + gus.positionsPreviouslyOccupied);

gus.move(new Position(3,1));
gus.eat(new Fruit(GameColors.RED));
gus.move(new Position(2,1));
gus.move(new Position(1,1));
gus.eat(new Fruit(GameColors.YELLOW));


System.out.println("\n2) Gus: " + gus);
System.out.println("Stack of previously occupied positions: " + gus.positionsPreviouslyOccupied);

gus.move(new Position(1,2));
gus.eat(new IceCream());

System.out.println("\n3) Gus: " + gus);
System.out.println("Stack of previously occupied positions: " + gus.positionsPreviouslyOccupied);

gus.move(new Position(3,1));
gus.move(new Position(3,2));
gus.eat(new Fruit(GameColors.ORANGE));


System.out.println("\n4) Gus: " + gus);
System.out.println("Stack of previously occupied positions: " + gus.positionsPreviouslyOccupied);

gus.move(new Position(2,2));
gus.eat(new SwissCheese());

System.out.println("\n5) Gus: " + gus);
System.out.println("Stack of previously occupied positions: " + gus.positionsPreviouslyOccupied);

gus.move(new Position(2, 3));
gus.eat(new Cake(4));
```

```
System.out.println("\n6) Gus: " + gus);
System.out.println("Stack of previously occupied positions: " + gus.positionsPreviouslyOccupied);
```

When executed the following will be displayed:



```
1) Gus: (3,2)
Stack of previously occupied positions: []

2) Gus: (3,1) (2,1) (1,1)
Stack of previously occupied positions: [(3,2)]

3) Gus: (1,2) (1,1) (2,1)
Stack of previously occupied positions: []

4) Gus: (1,1) (2,1) (3,1) (3,2)
Stack of previously occupied positions: [(1,2)]

5) Gus: (3,2) (2,2)
Stack of previously occupied positions: [(1,2), (1,1), (2,1), (3,1)]

6) Gus: (1,1) (2,1) (3,1) (3,2) (2,2) (2,3)
Stack of previously occupied positions: [(1,2)]
```