Assignment 2

COMP 250　　　　Fall 2025

Posted on:	Tuesday, Oct. 14, 2025
Suggested Deadline:	Tuesday, Nov. 4, 2025 at 23:59

Learning Objectives

In this assignment you will get some experience working with linked lists and stacks. You will implement a data structure to represent a caterpillar. This data structure is implemented as a singly linked list of segments.

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.

Lastly, this assignment will also give you more practice programming in Java! Although COMP 250 is not a course about how to program, programming *is* a core part of computer science and the more practice you get, the better you will become at it.

General Instructions

- **Submission instructions**

  - We encourage you to start early and aim to meet the suggested deadline (Nov. 4th), which is designed to be reasonable under standard circumstances. Re-submissions will be allowed until December 10th, but to receive Mastery you must submit a Proficient version of the assignment by the suggested deadline. Each time you submit, the results of some of the tests will be exposed, and you will know immediately what competency level you have achieved. Note that questions during office hours and on the discussion board will be prioritized based on upcoming deadlines.

  - 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**. The level of mastery you achieve with your latest submission is what will be considered when computing your final letter grade.

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

* Caterpillar.java

Do not submit any other files, especially .class files. Any deviation from these requirements may lead to lost marks.

– **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 or fields 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`.

- The assignment shall be graded automatically. Requests to evaluate the assignment manually shall not be entertained, so please make sure that you follow the instruction closely or your code may fail to pass the automatic tests. Note that for this assignment, you are NOT allowed to import any other class beside those already imported in the code provided. **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 certain exposed tests along with the competency level you've achieved. A small subset of these tests will also be shared with you to help with debugging. We highly encourage you to write your own tests and thoroughly test your code before submitting your final version. Learning how to test and debug your code is a fundamental skill to develop.

  You are welcome to share your tester code with other students on Ed and collaborate with others in developing it.

- Your submission will receive an "Inconclusive" if the code does not compile.

- Failure to comply with any of these rules may result in penalties. If anything is unclear, it is your responsibility to seek clarification, either by asking during office hours or posting your question on the Ed.

- IMPORTANT: Do NOT wait until you've finished writing the entire assignment to start testing your code. Debugging will be extremely difficult if you do so. If you need help with debugging, feel free to reach out to the teaching staff. When asking for help, be sure to mention:

  – The bug you're trying to fix.

  – What steps you've already taken to resolve it.

  – Where you've isolated the error.

Introduction[1]

Once upon a time, in a lush meadow nestled between the rolling hills of Munchgill Valley, lived a peculiar caterpillar named *Gus*. Gus had two defining traits: an adventurous spirit and a bottomless appetite.

One morning, as they inched along the grass, Gus discovered a magical patch brimming with endless treats—from strawberries and Swiss cheese to mysterious, shimmering snacks. Each bite had a curious effect: sometimes Gus grew longer, sometimes shrank, sometimes even reversed direction or shed a few segments. Odd? Yes. Delightful? Absolutely.

At the nearby Munchgill Valley University of Magic and Computer Science, a group of bright students noticed that Gus's antics looked exactly like updates to a data structure—changes applied one segment at a time. They decided to turn that insight into a game: model Gus's transformations with **singly linked lists**.

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'.

Your task is to represent Gus as a chain of nodes and implement the effects of different foods as list operations—adding and removing segments, reversing portions of the list, truncating or shedding tails, and performing other carefully defined updates. As you bring Gus's adventures to life, you'll practice reasoning about correctness, edge cases, and efficiency—one bite (and one node) at a time.

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 as follows:

- `assignment2` - This package contains the class `Caterpillar.java` that you need to modify in order to complete this assignment. You will need to understand and use the other classes contained in this package, but you should not modify them!

- `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 will be represented as a singly linked lists of `Segment`s. As you can see from the definition of the nested class `Segment` inside *Caterpillar.java*, each `Segment` stores the following information:

---

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

- A `Position`, representing where on the board this segment is. The position is completely defined by the x and y coordinates (as you can see from *Position.java*) and throughout this pdf we will represent such objects using the ordered pair (`x, y`).

- 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 `Segment`, representing the next segment in the body of the caterpillar. If this is the last body segment, then this field will be `null`.

Each caterpillar is defined by the following fields:

- `Segment head` : a reference to the head of the caterpillar

- `Segment tail` : a reference to the last segment in the caterpillar's body

- `int length` : the number of segments in the caterpillar's body

- `EvolutionStage stage` : the stage in which the caterpillar finds itself. `EvolutionStage` is an enumeration. 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.

- `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:

<center>(2,2) (2,1) (3,1)</center>

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.

- As also indicated in the java files, you will see that the all the fields of the Caterpillar class as well as the nested class Segment 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!

  - The goal will always be initialized with a value greater than 1.

  - None of the methods will be called if the caterpillar is in a butterfly or entangled stage.

Step 1: Growing the Basics

In this section, you'll start by laying down the groundwork for your caterpillar, Gus. The tasks here will focus on implementing the foundational methods that allow Gus to grow, move, and interact with its surroundings. These basic operations are essential for understanding how linked lists function in Java, and they will provide you with the building blocks to handle more complex tasks later on. Take your time understanding how to implement this part carefully!

- `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.
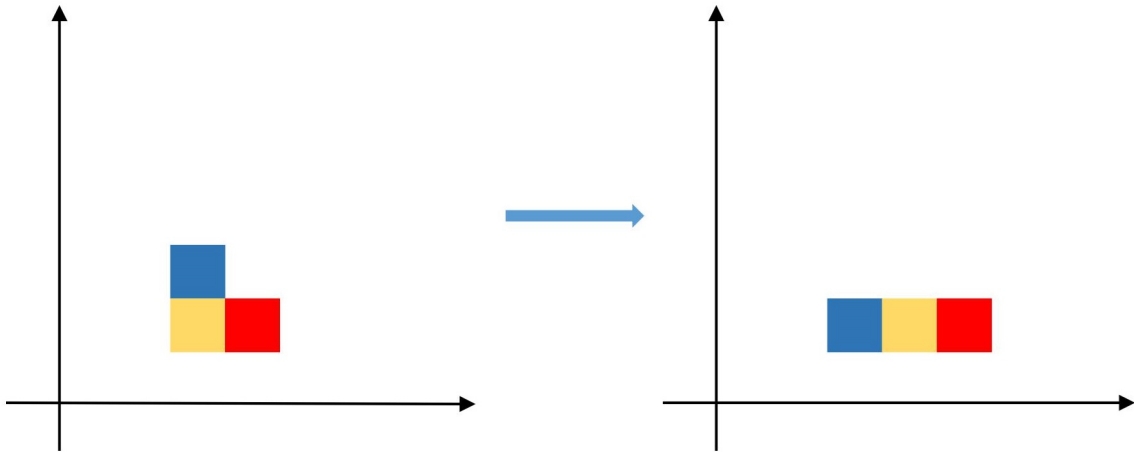
  This method runs in $O(1)$.

- `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)`).

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

Step 2: Tricky Treats

Now that Gus is up and running, it's time to deal with the more peculiar foods he encounters on its journey. In this section, you'll implement methods to handle the quirky consequences of Gus eating pickles, lollipops, and ice cream. Each food brings its own challenge, requiring you to manipulate Gus's body in more complex ways.

- `eat(Pickle p)` : the sourness of the pickle makes the caterpillar retrace its steps. This method updates its segments accordingly. Note that the position where the pickle was found should not be added to the stack of previously occupied positions.
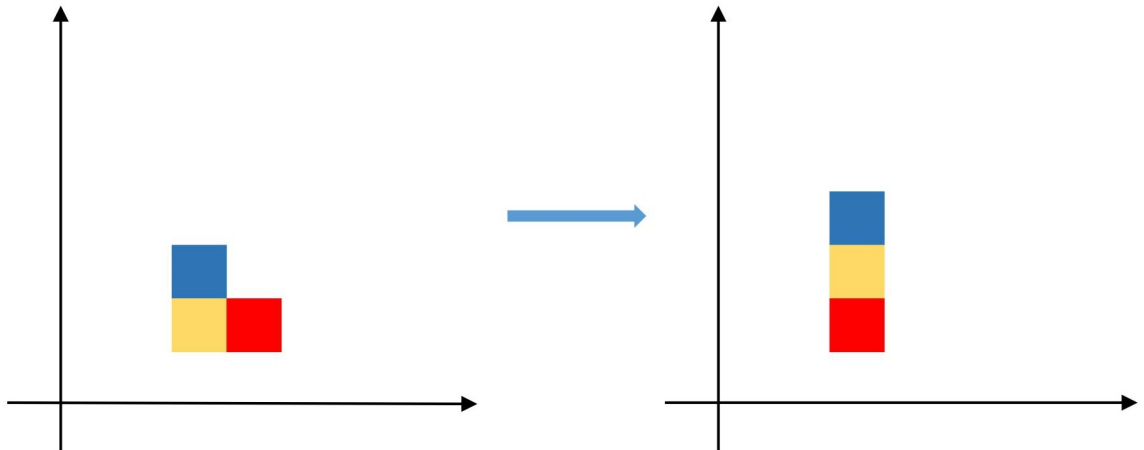
  For instance, suppose we have the following caterpillar:

  (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! In shock, its body does a full reversal, with its tail becoming the new head. Its (new) head turns `GameColors.BLUE` like an icicle, but all other segments retain their original colors. At this point, the caterpillar loses 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.

Step 3: Gus's Greatest Challenges

In this final section, you'll face Gus's toughest challenges yet. These tasks are designed to test your understanding of how linked lists work, especially when dealing with growth and transformation. From managing how Gus digests a cake and grows by multiple segments to handling the cheesy

delight that makes him shrink, you'll need to think critically and design your solutions with efficiency in mind. This section will push you to integrate all the skills you've learned so far, tackling complex operations with precision.
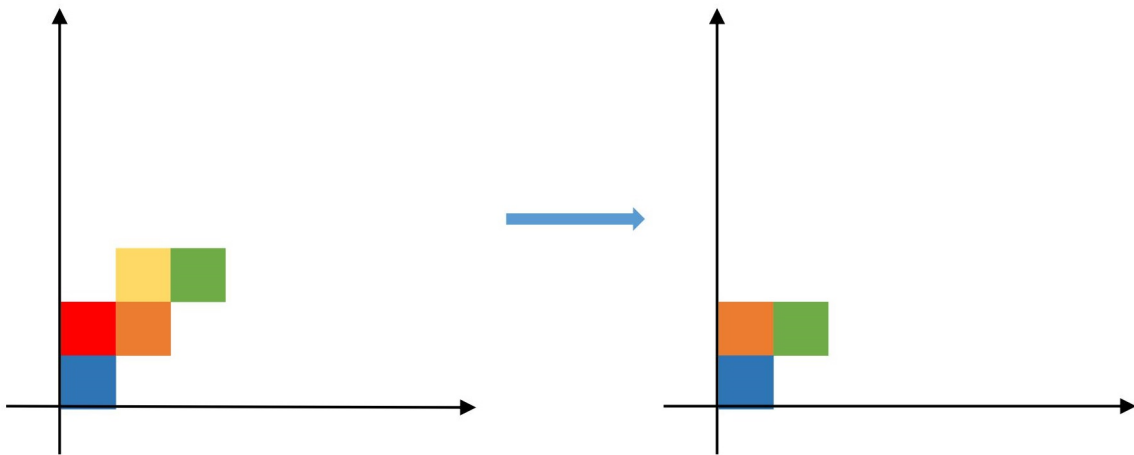
- `eat(SwissCheese cheese)` : who can resist the cheesy delight of those perfectly shaped eyes![2] But wait...by indulging on those cheesy holes the caterpillar body is getting "holey" too. This method shrinks the caterpillar who loses every other segment of the its body. This means that the segments 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 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:

$$\text{(2,2) (1,2) (1,1) (0,1) (0,0)}$$

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

$$\text{(1,1) (0,1) (0,0)}$$

  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 `[(2,2), (1,2)]`, with `(1,2)` 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

---

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

previously occupied positions that are still available to the caterpillar. For example, consider the following caterpillar:

<p style="text-align:center">(0,1) (0,0) (1,0)</p>

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)` : Time to go back to one of the first methods you implemented and add some functionality to it. 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)]
```