CS 245 - Project 2 Fall, 2025

Project 1 - Massive / Movement

The goal of this assignment is to demonstrate your mastery of the List data structure and all its variants, to demonstrate your understanding of lists and polymorphism and to code to an API in order to create a simulation of a star system similar to the one in this Zoom link (Passcode: S.?jY2wu).

Get started early. Avoid submitting late.

Background

There are many online tutorials for implementing 2-dimensional animations in Java. Using Java Swing and Java AWT, the two listed here allow you to:

- Create basic shapes[1]
- Create simple animations

These two tutorials may be a sufficient primer for performing the tasks required by this assignment.

You will use Java animations to simulate a number of celestial bodies (stars and comets?) on a two-dimensional canvas.

Requirements

Your implementation must satisfy a number of requirements, including each of the following:

- Reading a property file
- Creating List realisations
- Creating a canvas
- Creating and maintaining a List of celestial objects — one “star” and several random “comets”

These individual requirements are detailed below.

---

[1] I had to alter the sequence of statements in this video to the below in order to get it working:

```
Tutorial t = new Tutorial();
JFrame jf = new JFrame();
jf.setTitle("Roll Bounce");
jf.setSize(t.maxX, t.maxY);  // Window size defined in the class
jf.add(t);  // This appears below "setVisible" in the video
jf.setVisible(true);
jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Key	***Note(s)***
`timer_delay`	Canvas update frequency.
`list`	All lists used under this configuration must be the type specified. Valid types are: ● arraylist = ArrayList ● single = Singly-linked list with “next” pointers ● double = Linked list with “next” and “prev” pointers ● dummyhead = Linked list with empty “head” node
`window_size_x` `window_size_y`	Width (x) and height (y) of the canvas

`star_position_x`
`star_position_y`
`star_size`
`star_mass`
`star_velocity_x`
`star_velocity_y`
	The first celestial body on the canvas is a “star” with certain characteristics — x & y position on the canvas, radius (size) and x & y velocity in pixels per update. Your implementation may ignore the mass of the star in kg.

`gen_x` `gen_y`	Probability that a new celestial body will be generated at the frontier of the “x” axis (gen_x) and from the frontier of the “y” axis. If generated, the new celestial body has a 50% probability of being generated from either side (i.e. from the top / bottom or from the left / right).
`body_size`	Radius of the new celestial body.
`body_velocity`	The range of velocities of the celestial body. If the value is 3, the range of velocities must be random between -3 — +3, excluding 0. Values are determined separately for velocity along the x-axis as well as along the y axis.

Table 1: Selected keys and explanations

Requirement 1: Reading a property file

Your implementation is required to read the property file containing the details of implementation. The name of the property file will be provided as the first argument to your implementation. For example, if the name of your assignment is `MassiveMotion.java` and the property file is named `MassiveMotion.txt`, the implementation will be executed from command line as follows:

```
java MassiveMovement MassiveMovement.txt
```

Your implementation must use an instance of the `java.util.Properties` class (or another package approved by the instructor) to read the property file and subsequently to get the values stored in the property file. There are many examples of how to use the Properties class, including at Kode Java. A

sample property set of key-value pairs appears below. If the property file is missing, you may use the values below as defaults or you may abandon execution.

```
timer_delay = 75

list = arraylist

window_size_x = 1024
window_size_y = 768

star_position_x = 512
star_position_y = 384
star_size = 30
star_mass = 2E29
star_velocity_x = 0
star_velocity_y = 0

gen_x = 0.06
gen_y = 0.06
body_size = 10
body_mass = 1E21
body_velocity = 3
```

An explanation of some of the keys appears in Table 1. All “mass” keys (`body_mass` and `star_mass`), along with items not specified in Table 1, may be ignored.

Requirement 2: Creating List realisations

Your submission is required to show at least the four realisations (implementations) of the List interface enumerated in Table 2. Your submission must not use implementations from Java or other libraries — i.e. you must provide classes in the same package of these realisations. You may notice that the contents of Table 2 are purposefully in harmony with the List interface discussed in class. The List interface appears in the starter code for this project. Your implementation is required to implement the functions found in this interface. You may also add to this interface if you wish.

Class Name	**Value in property file**	**Note(s)**
ArrayList	`arraylist`	Array-based list with initial capacity of 10.
LinkedLIst	`single`	Linked list containing Node with `next` pointer.
DoublyLinkedList	`double`	Linked list containing Node with `next` and `prev` pointers.
DummyHeadLinkedList	`dummyhead`	LinkedList with `null` head Node.

Table 2: Required List implementations

Requirement 3: Creating a canvas

Your implementation is required to create a canvas of the height indicated by the property `window_size_y` and width indicated by the property `window_size_x` in pixels. The YouTube tutorials linked above indicate how to create a canvas of the appropriate size.

Requirement 4: Creating and maintaining a List of celestial objects

Your implementation is required to create a number of celestial objects ("`Graphics.fillOval()`"), initially at the edges of the canvas. All the celestial objects created by your implementation must be maintained in a single List realisation as defined in the starter code for the repository.

Your implementation may consider the first celestial object a star and paint it Color.RED as is shown in the example in the zoom link. The star's initial position (`star_position_x` and `star_position_y`), size (`star_size`) and velocity (`star_velocity_x` and `star_velocity_y`) in pixels per update are given by keys in the property file. The star must appear in a List of celestial objects.

The number of celestial objects must increase at each epoch of time. An epoch is defined by the value associated with timer_delay key and passed to the constructor of the Timer. A celestial object is created with the probability given by the keys `gen_x` and `gen_y`. The probability of generating one celestial object along the top or bottom of the canvas is given by the `gen_x` key, and the probability of generating one celestial object along the left or right side of the canvas is given by the `gen_y` key. The size of each celestial object is indicated by the `body_size` property. As celestial objects move beyond the canvas, these objects must be removed from the list which maintains them. Your implementation may paint each of these objects Color.BLACK.

Each celestial object moves a certain distance in pixels with each update. This distance is specified by the velocity keys `star_velocity_x` and `star_velocity_y` for the star and a randomly-generated value in the range of `-body_velocity` .. `+body_velocity` (excluding 0) for the comets. The changes to the celestial body locations are called or performed by the `actionPerformed(...)` function.

A note on code style

Your implementation must use self-documenting names for variables and functions. You may choose either camel case or snake case as long as you use one style consistently. Your implementation must use Javadoc comments — example below — for each (public) function. It may also be useful to use in-line comments for code doing something that is noteworthy or something that is not obvious. Examples of these are shown in the "fibonacci" function below.

```
/** fibonacci method recursively calculates and returns
  * the fibonacci value of a given integer.
  * @param integer n to calculate
  * @return integer fibonacci value at n
  */
int fibonacci(int n){
  if (n <= 1) return n;  // Base case
  else return fib(n-1) + fib(n-2);
 }
```

Submission

Accept this GitHub Classroom Assignment. Check your Java code — including any new class implementations, functions and interfaces — into the GitHub repository. Do NOT upload any .class files. To the README.md file in your repository, add descriptions, links to running implementations (such as the Zoom video above), etc.

Grading

Category	**Example questions**
Implementation	Other than visual verification, how did you determine the correctness of the implementation?
Efficiency	What is the running time of your List implementations functions? Which of these classes is optimal for this problem?
	How do you ensure that your simulation runs efficiently, especially if the number of celestial bodies increases?
	Have you implemented any performance optimizations?
	Are there any areas where you think your code might become inefficient as the number of celestial bodies increases?
Decomposition	What was your thinking in breaking the problem into smaller, manageable components? / Describe any new classes and the motivation behind them. / How did you decide on the responsibilities of each class?
	How would you extend this implementation to account for gravity? How would you extend this implementation to account for “crashes” of celestial bodies?

Table 3: Possible questions for Code Review

As with Project 1, grading will take place in two phases:

1. Once you submit your implementation, an evaluator will determine whether your implementation passes correctness and other quality standards, as detailed in the bullets below. If your implementation meets or exceeds these standards, the evaluator will mark your submission as “Reviewable.” You must sign up for a code review over zoom within 2 weeks of your implementation being marked “Reviewable.” Elements of quality standards are as follows:
    - Style: in the eyes of the evaluator, your implementation should be well-commented and must consistently use intelligently-named variables and functions.
    - Decomposition: in the eyes of the evaluator, your implementation must demonstrate a reasonable object oriented decomposition — i.e. encapsulation, polymorphism and inheritance.
    - Implementation: your class implementations must run successfully with valid configurations. The implementations must produce the expected results and smooth animations.
    - Efficiency: in the eyes of the evaluator, your implementation must be maximally efficient with respect to expected running time and required space.
    - Documentation: in the eyes of the evaluator, the github repository must be well-documented.
    - Code review: you will sign up for a 15-minute code review session to go over the quality of your code. The evaluator will use the rubric below. In some cases, your evaluator will be unable to provide a grade until after your code review. Note that the evaluator may ask questions in addition to what is listed.
2. During the 15-minute code review session, the evaluator will review the quality of your code with you via zoom. The evaluator will use the rubric below to ask you questions about your implementation. In some cases, your evaluator will be unable to provide a grade until after the review has been confirmed by a second evaluator.

Your grade for this project will be determined as follows:

<table>
<tr><th></th><th>Style</th><th>Decomposition</th><th>Implementation</th><th>Efficiency</th><th>Documentation</th><th>Code Review</th></tr>
<tr><td>Exemplary</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td></tr>
<tr><td>Satisfactory</td><td></td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td></tr>
<tr><td>Partial</td><td></td><td></td><td colspan="4">Submission not pass one or more of these</td></tr>
<tr><td>Unassessable</td><td colspan="6">Submission is missing, does not compile or does not work properly</td></tr>
</table>