C212 Final Exam (150 points)
April 29, 2024

Please read these directions before starting your exam.

This is a closed-note exam aside from your one page of notes, double-sided. You may not use any electronic devices to complete this exam, nor can you communicate with anyone besides the proctors and professor. *If you are caught cheating, you will receive an F in the course.*

For any question, unless specified otherwise, you may use any class without a corresponding `import`. E.g., if you want to use `HashMap`, you do not need to also import `java.util.HashMap`.

Unless otherwise stated, you do not need to spell out the "full design recipe", i.e., write the signature, documentation comments, and tests. Of course, doing so may aid you in your solution.

If you find a mistake, please raise your hand and let one of the proctors know; we will determine whether or not this is the case.

If you need to use the restroom, raise your hand and let a proctor know. You must turn in your exam, cheat sheet, and phone before leaving. You will receive these back upon your return.

When you are finished, turn in your exam and notes sheet if you have one, then quietly exit.

You have 120 minutes to complete the exam.

Good luck!

Question	Points	Score
1	60	
2	20	
3	30	
4	20	
5	20	
Total:	150	

Name: ____

IU Email: ____

Part I

Recommended Time: 60 minutes

2 Problems

1. (60 points) A *particle system* is a data structure that manages particles, or small effects, in a graphical engine. Think of a video game that has smoke, fire, water, explosion, or other kinds of effects. In general, these all use particle engines for managing hundreds of thousands of particle objects. Therefore, such an engine should be efficient. In this question, you will implement a particle system similar to one that I wrote a while ago!

   (a) *(4 points)* First, design the `Particle` class. A `Particle` contains a `double x` and `double y` representing its position, a `double width` and `double height` representing its dimensions, and a `double dx` and `double dy` representing its velocity. Finally, it contains a `double life` representing its life. The constructor should receive these as parameters and assign them to the instance variables. You do **not** need to write the respective accessors and mutators, and for all future problems, you may assume they are trivially defined.

   (b) *(4 points)* Inside the `Particle` class, design the `update` method, which adds the particle's velocity to its position. It should also decrement the `life` instance variable by one. If `life` ever becomes zero or negative, the particle is no longer alive. If the particle *isn't* alive, do not update its position (nor decrement its life).

   (c) *(2 points)* Design the `isAlive` method that returns whether or not the particle is alive.

   The skeleton code is on the next page.

```
class Particle {








  Particle(_________________________________________________________________) {













  }

  /**
   *
   */
  _____________ update() {
















  }

  _____________ isAlive() {



  }
}
```

The idea behind this particle system is that we create a *memory pool*, and poll already-allocated particles from it when available. That is, when a particle dies, it moves to the "dead" sector, but that memory still exists. Then, when we want to create a new `Particle`, we first check to see if there are any dead particles that we can reuse. If so, we reuse that particle's allocated memory and simply reassign variables.

(d) *(5 points)* Design the `ParticleSystem` class. Store the following instance variables and instantiate them as `LinkedList` instances in the constructor. The constructor should also receive a value `maxAlive`, which is assigned to a `final int MAX_ALIVE` instance variable.

- `List<Particle> alive`, which stores the alive particles in the system. All particles in this list should be non-`null`.
- `List<Particle> dead`, which stores the dead particles in the system. All particles in this list should be non-`null`.

(e) *(20 points)* Design the `boolean addParticle(double x, double y, double w, double h, double dx, double dy, double life)` method that adds a particle to the system with the given parameters. If there are no dead particles available, then simply allocate a `new Particle` onto the rear of the `alive` list. If there is a dead particle, use that allocated space instead and assign the parameters to the object using the respective setters. Then, move the particle out of the `dead` list and onto the rear of the `alive` list. If it is impossible to add a new particle (because there is no space for more alive particles), return `false`. Otherwise, return `true`.

The skeleton code is on the next page.

```
class ParticleSystem {
  // ... previous methods not shown.

  /**
   *
   *
   * @param x -
   * @param y -
   * @param w -
   * @param h -
   * @param dx -
   * @param dy -
   * @param life -
   * @return
   */
   boolean addParticle(double x, double y, double w, double h,
                       double dx, double dy, double life) {
     // First check to see if there's room anywhere in the system.
     if (_______________________________) {
       return false;
     } else {
       // There must be room, so let's determine if there's a dead particle.
       Particle p = null;
       if (______________________________) {
         // Remove the first particle off the front of "dead".
         p = ______________________________;
         // Update the fields of p to those given as parameters.
         p.set______(__________);
         p.set______(__________);
         p.set______(__________);
         p.set______(__________);
         p.set______(__________);
         p.set______(__________);
         p.set______(__________);
         p.setAlive(true);
       } else {
         // Just allocate space for the new particle.
         p = ______________________________________________________;
       }
       // Add p onto the rear of the alive list.
       ____________________________________
       // Return success.
       return ________;
    }
  }
}
```

(f) *(2 points)* Design the `void removeParticle(Particle p)` method that *prompts* for `p` to be removed from the "alive" queue. All this method should do is toggle *p* to be no longer alive. By *prompt*, we mean that it does not affect either `List`.

```
class ParticleSystem {
  // ... previous methods not shown.

 /**
  *
  * @param p -
  */
  void removeParticle(Particle p) {


  }
}
```

(g) *(8 points)* Design the `void updateSystem()` method that traverses over the alive particles, and invokes their `update` methods. After invoking a particle's `update` method, check to see if it is alive or not. If it is not alive, move it out of the `alive` list and into the `dead` list.

```
class ParticleSystem {
  // ... previous methods not shown.

  /**
   *
   *
   */
  void updateSystem() {
    for (int i = 0; i < _____________________; i++) {
      // If the particle is dead, remove it and add to dead list.
      if (______________________) {
        this._________.add(this.___________.remove(_______));
        _______________;
      } else {
        // Otherwise, update the particle.
        ________________________________;
      }
    }
  }
}
```

Answer the following questions with at most 2-3 sentences. *Do not throw everything and the kitchen sink into your answer!*

(h) *(10 points)* Why do we not traverse the `alive` list inside `removeParticle` to remove the given particle directly? What are the performance implications of doing so?

(i) *(10 points)* The particle system knows the maximum capacity of alive and dead particles. Despite this, we still choose to use a dynamically-allocated list for storing references to the alive and dead particles. It may seem like a better choice to use an array instead, and simply use one-half for the alive particles and one-half for the dead particles. What is the **MAIN** disadvantage of this approach? Hint: what can we not do with arrays that we can with lists?

2. (20 points) This question has two parts and reuses the `Particle` class from the first question.

   (a) *(10 points)* Design the `SparkParticle` class, which inherits from `Particle`. "Spark particles" move in a straight line, but their velocity decreases over time due to air resistance until they stop moving.

   - The `SparkParticle` constructor receives the same values as its superclass counterpart.
   - Override the `update` method to decrease the vertical and horizontal velocities by 10% with each call to `update`. Do *not* call `super.update()`. Instead, update the position of the particle directly inside this class. Remember that those variables are private in the `Particle` class.
   - Override the `isAlive` method to return `false` when its horizontal and vertical velocity values are both less than 0.01 away from zero. Otherwise, it should return `true`.

(b) *(10 points)* Design the `SmokeParticle` class, which inherits from `Particle`. "Smoke particles" move in a straight line, but their velocity decreases over time due to air resistance until they stop moving.

- The `SmokeParticle` constructor receives the same values as its superclass counterpart.
- Override the `update` method to increase the width and height dimensions by 2% with each call to `update`. Do *not* call `super.update()`. Instead, update the position of the particle directly inside this class (the behavior is the same as the `Particle` superclass. Remember that those variables are private in the `Particle` class. Finally, decrement the life by `0.2` rather than `1`.

Part II

Recommended Time: 60 minutes

3 Problems

3. (30 points) This question has two parts.

   (a) *(15 points)* Design the `cycleOperationsTR` and `cycleOperationsTRHelper` methods to circularly apply `+` then `*` to the elements of the list. The former acts as the driver to the latter; the latter solves the same problem that `cycleOperations` does, but it instead uses tail recursion. Remember to include the relevant access modifiers! Use the following examples to guide your design.

```
cycleOperations({})
    => 0
cycleOperations({1, -4, 9, -16, 25, -36})
    => 0 + (1 - (-4 * (9 + (-16 - (25 * -36)))))
    => 1980
cycleOperations({10, 5, -2, 1, -2})
    => (((((0 + 10) * 5) + -2) * 1) * -2
    => 46
```

(b) *(15 points)* Design the `cycleOperationsLoop` method, which solves the problem using either a `while` or `for` loop.

4. (20 points) Design the `MapSumPairs` class that supports two operations: `void insert(String s, int v)`, and `int sum(String suffix)`. The former adds the association of `s` to `v` in a map. The latter returns the sum of all values whose keys end with the given suffix. You must follow the "design recipe" laid out in class. That is, you must write the method purpose statements, tests, and the implementation. You may write your tests as a series of insert calls, followed by calls to `sum`.

   The tester skeleton code is on the next page, and the class skeleton is on the page thereafter.

```
import static Assertions.assertAll;
import static Assertions.assertEquals;

class MapSumPairsTester {

  @Test
  void testMapSumPairs() {
    MapSumPair m1 = new MapSumPair();
    m1.insert(_______, _______);
    m1.insert(_______, _______);
    m1.insert(_______, _______);
    m1.insert(_______, _______);

    assertEquals(_______________________, m1.sum(_____________))
    assertEquals(_______________________, m1.sum(_____________))
    assertEquals(_______________________, m1.sum(_____________))

    MapSumPair m2 = new MapSumPair();
    m2.insert(_______, _______);
    m2.insert(_______, _______);
    m2.insert(_______, _______);
    m2.insert(_______, _______);

    assertEquals(_______________________, m2.sum(_____________))
    assertEquals(_______________________, m2.sum(_____________))
    assertEquals(_______________________, m2.sum(_____________))
  }
}
```

```
import java.util.*; // Import all necessary collections.

class MapSumPair {

}
```

5. (20 points) Oh no! Janmejay's rabbit, Oreo, has nibbled part of this exam away and we need you to fix the missing code. Fill in the missing code for this insertion sort implementation. Note that this is a *functional* implementation of the insertion sort, which means that we return a new list rather than sorting the one we provide.

```
import java.util.List;

interface IInsertionSort<___________________> {
  List<___> insertionSort(List<___> ls);
}


class FunctionalInsertionSort<__________________> implements _______________ {

  @Override
  public List<___> insertionSort(List<V> ls) {
    if (ls.isEmpty()) { return _____________________ }
    else {
      return insert(__________,
                    _________________________________________________);
    }
  }

  /**
   * Inserts an element into a sorted list of values.
   * @param val - value to insert.
   * @param sortedRest - a sorted sublist.
   * @return the sorted sublist with the new value inserted.
   */
  private List<___> insert(___ val, List<___> sortedRest) {
    if (sortedRest.isEmpty()) {
      ________<___> ls = new ArrayList<>();
      ls.add(____);
      return _____;
    } else if (_______________________________ < 0) {
      List<V> ls = new ArrayList<>();
      ls._____(______);
      ls.addAll(____________);
      return _____;
    } else {
      _________<___> ls = new ArrayList<>();
      ls.add(________________);
      ls.addAll(_________________________________________________________)
      return _____;
    }
  }
}
```

Scratch work