C212 Final Exam (150 points)
April 29, 2024

C212 Final Exam Rubric

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.

Solution.

Rubric:

- (a) All instance variables are present. +2
- (a) The constructor exists. +1
- (a) All values are correctly initialized. (+1)
- (b) Velocity is updated. (+2)
- (b) Particle is not updated if dead. (+1)
- (b) Particle's life is decremented. (+1)
- (c) `isAlive` is correct. (+2)

```
class Particle {

  private double life, x, y, width, height, dx, dy;

  Particle (double x, double y, double width, double height, double dx, double dy, double life) {
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
    this.dx = dx;
    this.dy = dy;
    this.life = life;
  }

  /**
   * Some comment...
   */
  void update() {
    if (this.life <= 0) { return; }
    else {
      this.x += this.dx;
      this.y += this.dy;
      this.life--;
    }
  }

  boolean isAlive() {
    return this.life > 0;
  }
}
```

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`.

Solution.
Rubric:

- All three instance variables exist. (+2.5).
- `MAX_ALIVE` is final. (+0.5).
- All variables are correctly assigned and instantiated as instructed. (+2)

```
class ParticleSystem {

  private List<Particle> alive;
  private List<Particle> dead;
  private final int MAX_ALIVE;

  ParticleSystem(int maxAlive) {
    this.alive = new LinkedList<>();
    this.dead = new LinkedList<>();
    this.MAX_ALIVE = maxAlive;
  }
}
```

(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.

Solution.

Rubric:

- Javadoc exists (+3).
- Size check (+2).
- `if (!this.dead.isEmpty())` (+2)
- Removes first particle (+3)
- Correctly copies variables over USING SETTERS (+6). Just assigning them directly awards ONLY ONE POINT.
- Correctly calls constructor (+2).
- Adds particle onto end of alivel ist. (+1)
- Returns true. (+1)

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

  /**
   * ... some huge javadoc
   */
  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 (this.alive.size() >= MAX_ALIVE) {
      return false;
    } else {
      // There must be room, so let's determine if there's a dead particle.
      Particle p = null;
      if (!this.dead.isEmpty()) {
        // Remove the first particle off the front of "dead".
        p = this.dead.removeFirst();
        // Update the fields of p to those given as parameters.
        p.setX(x);
        p.setY(y);
        p.setWidth(w);
        p.setHeight(h);
        p.setDx(dx);
        p.setDy(dy);
        p.setLife(life);
      } else {
        // Just allocate space for the new particle.
        p = new Particle(x, y, w, h, dx, dy, life);
      }
      // Add p onto the rear of the alive list.
      this.alive.add(p);
      // Return success.
      return true;
    }
  }
}
```

(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`.

Solution.

Rubric:

- Uses setter (+2). Zero points if anything else. If they included an instance variable that sets the life, it can only be awarded points if it was declared in part (a).

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

 /**
  * Blah blah blah
  * @param p -
  */
  void removeParticle(Particle p) {
    p.setLife(0);
  }
}
```

(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.

Solution.

Rubric:

- Javadoc. (+1)
- Loop condition is correct. (+1)
- Correct `if` condition (+2).
- Correct `if` consequent body. (+3)
- Correct `if` alternative body. (+1)

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

  /**
   * Some comment...
   */
void updateSystem() {
  for (int i = 0; i < this.alive.size(); i++) {
    this.alive.get(i).update();
    if (!this.alive.get(i).isAlive()) {
      this.dead.add(this.alive.remove(i));
      i--;
    }
  }
}
```

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?

Solution. The problem is that this would result in a $\Theta(n^2)$ operation in the worst-case, where n is the number of (alive) particles in the system. One loop over each particle, then another for `removeParticle` if called.

(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?

Solution. Removing an alive particle would leave a slot ”open” and we’d have to keep track of those free slots when reallocating a particle. A list can dynamically resize; an array cannot.

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 or equal to 0.01 away from zero. Otherwise, it should return `true`.

   Solution.

   Rubric:

   - Constructor is correct (+3). If this class contains ANY instance variables, no points are awarded for this.
   - `update` is correct (+4).
   - `isAlive` is correct (+3).

```
class SparkParticle extends Particle {

  SparkParticle(double x, double y, double width, double height,
                double dx, double dy, double life) {
    super(x, y, width, height, dx, dy, life);
  }

  @Override
  public void update() {
    this.setX(this.getX() + this.getDx());
    this.setY(this.getY() + this.getDy());
    this.setDx(this.getDx() * 0.90);
    this.setDy(this.getDy() * 0.90);
  }

  @Override
  public boolean isAlive() {
    return this.getDx() <= 0.1 && this.getDy() <= 0.1;
  }
}
```

(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`.

Solution.

Rubric:

- Constructor is correct (+3). Same stipulations as the former.
- `update` is correct. (+7)

```
class SmokeParticle extends Particle {

  SmokeParticle(double x, double y, double width, double height,
                double dx, double dy, double life) {
    super(x, y, width, height, dx, dy, life);
  }

  @Override
  void update() {
    this.setWidth(this.getWidth() * 1.02);
    this.setHeight(this.getHeight() * 1.02);
    this.setX(this.getX() + this.getDx());
    this.setY(this.getY() + this.getDy());
    this.setLife(this.getLife() - 0.2);
  }
}
```

Part II

Recommended Time: 60 minutes

3 Problems

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

   Solution.

   Rubric:

   - Driver is correct (+2).
   - Base case is correct (+2).
   - Correct return value (+1).
   - Body is correct and correctly switches between the three operations using TR (+5). 1 pt for the operations. If it's not TR, 0 points total.

```
static int cycleOperationsTr(List<Integer> vals) {
  return cycleOperationsTrHelper(vals, 0, 0, 0);
}

private static int cycleOperationsTrHelper(List<Integer> vals, int idx, int c, int res) {
  if (idx >= vals.size()) {
    return res;
  } else {
    if (c % 2 == 0) {
      return cycleOperationsTrHelper(vals, idx + 1, (c + 1) % 2, res + vals.get(idx));
    } else {
      return cycleOperationsTrHelper(vals, idx + 1, (c + 1) % 2, res * vals.get(idx));
    }
  }
}
```

Solution.

Rubric:

- Loop condition is correct (+3).
- Value is correctly accumulated (+5).
- Correct return value (+2).

```
static int cycleOperationsLoop(List<Integer> vals) {
  int idx = 0;
  int c = 0;
  int res = 0;
  while (!(idx >= vals.size())) {
    if (c % 2 == 0) {
      res = res + vals.get(idx);
    } else {
      res = res * vals.get(idx);
    }
    idx = idx + 1;
    c = (c + 1) % 2;
  }
  return res;
}
```

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.

   Solution.

   Rubric:

   - (4 pts) at least two coherent examples.
   - (2 pts) sensible purpose statement.
   - (14 pts) definition works as expected.

```
class MapSumPairsTester {

  @Test
  void testMapSumPairs() {
    assertAll(
        Some sensible examples... :D
    );
  }
}

package problem4;

import java.util.HashMap;
import java.util.Map;

class MapSumPairs {

  private Map<String, Integer> M;

  MapSumPairs() {
    this.M = new HashMap<>();
  }

  void insert(String s, int n) {
    this.M.put(s, n);
  }

  int sum(String s) {
    int su = 0;
    for (String k : M.keySet()) {
      if (k.endsWith(s)) {
        su += M.get(k);
      }
    }
    return su;
  }
}
```

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.

Solution.

Rubric:

- −1 point for each incorrect blank up to −20.

```
import java.util.List;

interface IInsertionSort<T extends Comparable<T>> {
  List<T> insertionSort(List<T> ls);
}

class FunctionalInsertionSort<T extends Comparable<T>> implements IInsertionSort<T> {
  @Override
  public List<T> insertionSort(List<T> ls) {
    if (ls.isEmpty()) { return new ArrayList<>(); }
    else {
      List<T> rest = ls.subList(1, ls.size());
      return insert(ls.get(0), insertionSort(rest));
    }
  }

  private List<T> insert(T val, List<T> sortedRest) {
    if (sortedRest.isEmpty()) {
      List<T> ls = new ArrayList<>();
      ls.add(val);
      return ls;
    } else if (val.compareTo(sortedRest.get(0)) < 0) {
      List<T> ls = new ArrayList<>();
      ls.add(val);
      ls.addAll(sortedRest);
      return ls;
    } else {
      List<T> ls = new ArrayList<>();
      ls.add(sortedRest.get(0));
      ls.addAll(insert(val, sortedRest.subList(1, sortedRest.size())));
      return ls;
    }
  }
}
```