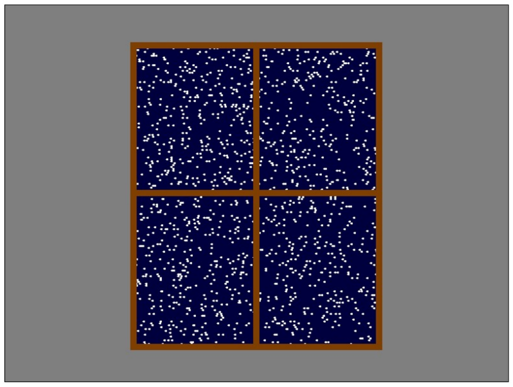
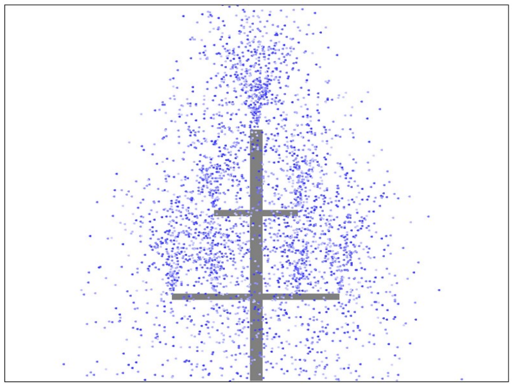
Particle Systems

Many computer-generated visual effects, whether in blockbuster Hollywood movies or in video games, make use of ***particle systems***. As the name suggests, a particle system is a system that consists of individual units called ***particles*** Those particles can have attributes like position, color, velocity, etc., and by maintaining systems consisting of hundreds or thousands of particles it's possible to build some beautiful effects from simple pieces.

In this part of the assignment, you'll implement a simple 2D particle system that can be used to create a variety of aesthetically-pleasing scenes. As a preview of where you're going, here are two of those scenes: a snowy day, and a water fountain:





Each of these scenes consist of some number of particles. Those are the colorful objects moving around the screen. The particle system is the overall system responsible for maintaining those particles, updating their positions in space, etc.

Your overall goal is to implement the following class:

```
public class ParticleSystem {
    public ParticleSystem() { … }

    public void add(Particle particle) { … }
    public int  numParticles() { … }
    public void drawParticles(ParticleGraphics g) { … }
    public void moveParticles() { … }
}
```

Let's walk through how this works. You're familiar with constructors and can probably guess what the top function does; `ParticleSystem` creates a new, empty particle system.

The next four functions are where the magic happens. The `add` function adds a new particle to the scene. It takes as input a `Particle` object, which contains information about the particle (where it is in space, which color it is, etc.). We'll describe the `Particle` type in more detail later on. The next function is `numParticles`, which says how many particles there are in the system. The `drawParticles` function, as the name suggests, actually draws the particles on the screen so that the user can see them in all their particular glory. Finally, there's `moveParticles`, which moves all of the particles a tiny bit in space. How those particles move depends on what kind of particles you're dealing with, and that's described later.

Rather than sharing the remaining details all at once, we'll incrementally introduce the different components of the system. We've broken this project down into several milestones so that you can slowly add features in one at a time.

Milestone One: Implement Core Functions

Your first task is to implement basic functionality for the `ParticleSystem` type. Specifically, you'll implement the constructor, `add`, and `numParticles` functions.

To provide sufficient context to work through this milestone, let's talk a bit more about the types referenced above. You may have noticed that the `ParticleSystem.add` function takes as input an object of type `Particle`, so let's start there. Each particle has some information associated with it. That information is stored in the `Particle` type defined in `Particle.java`. Here's what that looks like:

```
public class Particle {
    public double x, y;   // Position
    public Color color;   // Color

    /* ... four other fields we'll describe later. ... */
}
```

Let's walk through what these fields mean. The `x` and `y` fields tell you where in space the particle is. The coordinate system we use here is the same as for graphics: the upper-left corner of the scene is at (0, 0), with *x* increasing from left to right and *y* increasing from top to bottom. The `color` field, unsurprisingly, tells you what color the particle is.

When a client of `ParticleSystem` calls `add`, the particle they provide needs to be stored somewhere internally inside the `ParticleSystem`. To see how, let's look at the following fields and types of `ParticleSystem`:

```
static class Cell {
    public Particle particle;
    public Cell next, prev;
}

Cell head;
```

The `Cell` type represents a doubly-linked list cell that contains information about a particle. The `particle` field represents the actual particle. The two references `next` and `prev` point to the next and previous particles in the scene, respectively.

Inside the `ParticleSystem` type, the actual linked list of particle cells is pointed at by the `head` data member. This should initially be set to `null` if there are no particles, and otherwise should point to the first particle in the list.

Whenever you add a new particle to the system by calling `add`, that particle should be inserted at the ***back*** of the doubly-linked list. (There's a good reason for this that we'll describe later.) Your implementation of `add` should run in time O(1), though, which means that you'll need to find a good way to track where the end of the list is so that you can jump there without having to scan the entire list.

Finally, let's talk about the `numParticles` function. This function should return how many total particles are in the list. However, it must run in time O(1), meaning that the runtime of the function should be *independent* of the number of particles in the list.

To hit the time bounds for `add` and `numParticles`, you will need to add your own private member variables to `ParticleSystem`.

With that all being said, let's summarize what you need to do:

Milestone One Requirements

1. Implement the constructor for the `ParticleSystem` type. These should initialize a new particle system with no particles in it and clean up all the particles in the system, respectively.
2. Implement the `add` function. This function takes as input a `Particle` object, then creates a new `ParticleCell` and wires it onto the back of the linked list. This function needs to run in time O(1).
3. Implement the `numParticles` function. This function returns the number of particles in the system, and should do so in time O(1).
4. Add at least one - and possibly multiple - custom test cases. We recommend having those test cases check that the values of any member variables you introduce are correct.
5. Use our provided tests to make sure all the tests for this milestone are passing.

Some notes on this problem:

- In this part of the assignment - and more generally, throughout this assignment - you ***must not*** use any collection types (e.g. `ArrayList`, `HashMap`, `String`, etc.). After all, the purpose of this assignment is to help you get accustomed to working with linked lists, which necessitates thinking about data storage in a different way.
- ***Draw pictures!*** You don't need to write much code here, but the code you write will require a good deal of attention to detail about which objects link to which.
- You are welcome to add as many helper methods as you'd like to the `ParticleSystem` class.

Milestone Two: Draw Particles

You now have a rough outline of a particle system, but right now there's no way for anyone to see the particles you're storing. How sad – you've painted your masterpiece and then hidden it from the world! Let's rectify this.

Your next task is to implement the `ParticleSystem.drawParticles` function. As the name suggests, this function should draw all the particles that are in the system. You should draw them in the order in which they're stored in the particle system, so older particles (toward the front of the list) should be drawn before newer particles (toward the end of the list).

We've provided you with a type called `ParticleGraphics` that you should use to do all the drawing. There are two methods you will need to use, plus a bunch of others that are meant for later in the assignment. The relevant bits are here:

```
public class ParticleGraphics {
    public void setColor(Color color) { … }
    public void drawParticle(double x, double y) { … }
}
```

The `setColor` method changes the color of all subsequently-drawn objects. The `drawParticle` method takes as input an *x* and *y* coordinate and draws a particle there.

To recap, here's what you need to do:

Milestone Two Requirements

1. Implement the `ParticleSystem.drawParticles` function, which draws all the particles in the system in the order they're stored.
2. Run our provided test cases to confirm that everything is working just fine. (You don't need to write any test cases for this milestone.)

Some notes on this problem:

- You should implement this method iteratively rather than recursively, since when dealing with large particle systems there might not be enough stack space to recursively walk over all the particles.
- Our provided test cases make use of a custom subclass of `ParticleGraphics` called called `ParticleCatcher`. This is an object that intercepts calls to `drawParticle` so that instead of drawing the particles on the screen, the particles get written down for later inspection. We'll use this type more extensively in the later parts of this assignment as a way of checking that you've got all the particles in the right place.

Milestone Three: Move Particles

Your next task is to implement the `ParticleSystem.moveParticles` function, which moves all the particles in the system by a small amount. In order to do so, we're going to need to expose a bit more information about `Particle`. In addition to the `x`, `y`, and `color` fields you've seen thus far, the `Particle` type has these others:

```
public class Particle {
    public double x, y;   // Position
    public Color color;   // Color

    public double dx, dy; // Velocity

    /* ... two other fields we'll describe later. ... */
}
```

The `dx` and `dy` variables represent the velocity of the particle. Each time you call the `ParticleSystem`'s method `moveParticles`, each particle's `x` and `y` fields are increased by the value of its `dx` and `dy` values, respectively. So, for example, if a particle has `dx = 0` and `dy = 1`, then each time `moveParticles` is called the particle's `x` field will be unchanged and the particle's `y` field will increase by 1. This has the next effect of moving the particle down a little bit: its `x` position remains the same while its `y` position gets closer to the bottom of the screen. On the other hand, if a particle has `dx = 3` and `dy = -4`, then whenever `moveParticles` is called that particle's `x` coordinate will be increased by 3 and its `y` coordinate will be decreased by 4. This has the effect of moving the particle up and to the right. Finally, it's entirely possible that a particle has `dx` and `dy` set to zero, in which case it doesn't move.

Note that in this process the values of `dx` and `dy` for each particle don't change. Later on we'll revisit this and make it possible for particles to have changing velocities – but you don't need to worry about that now.

When you create a particle with `add`, the `Particle` you are provided will include a `dx` and `dy`, which you should store inside your `ParticleSystem`. Depending on how you wrote `add`, you may or may not need to edit `add` to make this happen. You'll then use the `dx` and `dy` values of each particle to move that particle in the `moveParticles` function.

Once you've done this, run our tests to confirm that everything is working correctly.

Here's a summary of what you need to do:

Milestone Three Requirements

1. Ensure that `ParticleSystem.add` correctly stores the `dx` and `dy` information for the provided particle. Depending on how you implemented `ParticleSystem.add`, this might not require you to write any new code.
2. Implement the `ParticleSystem.moveParticles` function. This method should move all the particles in the system by the amount specified in their `dx` and `dy` fields.
3. Run our provided tests to make sure all the provided tests for this milestone are passing.

Some notes on this problem:

- As with `drawParticles`, don't implement this one recursively. Iteration is your friend!
- The velocity of a particle can be arbitrarily large or small, and you shouldn't make any assumptions about the sign or magnitude of `dx` and `dy`.
- Each particle has its own velocity, which is independent of the velocities of all other particles.

Milestone Four: Cull Particles

So far, your particle system only has a mechanism for adding particles, and there's no way for particles to be removed. However, particles don't live forever. Specifically, there are two circumstances where particles can be removed.

First, you should remove a particle *if that particle goes out of bounds*. To make sure we aren't tracking particles that can't appear on the screen, when a particle is no longer on-screen, you should remove it. Specifically, a particle is considered off screen if its *x* or *y* coordinate is less than 0, if its *x* coordinate is greater than or equal to the special constant `SCENE_WIDTH`, or if its *y* coordinate is greater than or equal to the special constant `SCENE_HEIGHT`.

Second, you should remove a particle *if that particle's lifetime ends*. Inside of `Particle` is an `int` called `lifetime`, as shown here:

```
public class Particle {
    public double x, y;    // Position
    public Color color;    // Color

    public double dx, dy; // Velocity
    public int lifetime;  // How much longer the particle lives

    /* ... one other field we'll describe later. ... */
}
```

Here's how `lifetime` works. Each time a particle moves, its lifetime decreases by one. As soon as a particle's lifetime is negative (that is, not zero, and not positive), the particle ceases to exist. If a particle has a large lifetime, it will move for a very long time before disappearing (unless it goes out of bounds). If a particle has a small lifetime, it will only move a few times before disappearing. A particle's lifetime is initially specified by the value of the `Particle` passed into the `add` function based on the needs of whoever is using the particle system.

There are two places you need to check for particles that need to be removed, First, when `moveParticles` is called, a particle could either move offscreen or have its lifetime run out. In that case, as soon as your call to `moveParticles` realizes this, it should remove the particle. Second, it's possible that the particle's position was never onscreen to begin with when it was added in `add`, and similarly a particle might initially have a negative lifetime when `add` was called. Either way, your `add` method shouldn't add these particles in the first place.

You will need to make some edits to your code in order to support this functionality. The trickiest bit will be in your `moveParticles` code. Remember that rule is that ***as soon as a particle needs to be removed, you should remove it***. This means that you should just have one loop in your `moveParticles` function, which will loop over the particles, moving each particle and adjusting lifetimes, and immediately removing particles that are out of bounds or whose lifetimes end. You will need to be careful when doing this. In particular, you'll be removing particles from the linked list at the same time that you're iterating over that list. Pay close attention to how you advance from one particle to the next to make sure that you don't accidentally skip over particles, wire a loop into the list, etc. ***Draw lots of pictures*** as you do this; it will make it a lot easier to see what you need to do here.

This property of particle systems – that arbitrary particles might need to be removed from the system – is the reason why we're storing particles in a linked list. If we used, say, a regular dynamic array, then the cost of removing a single particle would be O(n) in the worst case, since we'd need to shift all the other particles back a spot. On the other hand, using a linked list allows us to remove a particle in time O(1) by just splicing it out of the list.

To recap, here's what you need to do:

Milestone Four Requirements

1. Update your code for `add` to ensure that you store particle lifetimes for later, that you don't add particles that are offscreen, and that you don't add particles with negative lifetimes.
2. Update your code for `moveParticles` to adjust lifetimes and remove particles whose lifetime becomes negative or that have moved offscreen. Remember that you must remove particles as soon as you discover that they need to be removed.
3. Run our provided tests to make sure all the provided tests for this milestone are passing.

Some notes on this part of the assignment:

- ***Draw pictures!*** The linked list manipulations required here are not exceedingly complex, but they aren't trivial either. Having a diagram of what pointers point to which cells will help you better understand what you need to do - and debug things if something goes wrong.
- Iterating over a list while removing items from it is tricky. Make sure to handle the case where the item you're removing is at the front of the list, the back of the list, and somewhere in the middle of the list. It's easy to introduce bugs into any of these places.
- Feel free to add as many helper methods as you'd like.

Milestone Five: Support Different Particle Types

You now have a working particle system – nicely done! Your final task is to make that particle system more interesting by supporting three different types of particles.

There is one final field in the `Particle` type that we haven't talked about yet. It's shown here:

```
public class Particle {
    public double x, y;    // Position
    public Color color;    // Color

    public double dx, dy; // Velocity
    public int lifetime;  // How much longer the particle lives

    ParticleType type;    // What kind of particle?
}
```

Here, `ParticleType` is an `enum` defined as follows:

```
public enum ParticleType {
    STREAMER, BALLISTIC, FIREWORK
}
```

These three options correspond to the three different types of particles you will need to support.

The first type is `ParticleType.STREAMER`, which is the sort of particle you have seen so far. It has a lifetime that drops each step and moves according to a fixed velocity.

The second type is `ParticleType.BALLISTIC`. This type of particle works exactly the same way as a streamer but with one change: ***every time a ballistic particle moves, after it moves its `dy` increases by one***. This tiny change means that ballistic particles feel the effects of gravity and move downward faster and faster on each frame. Ballistic particles are otherwise completely identical to streamers. (Those of you with a physics background might see why this will work, but physics is by no means necessary to code this one up!)

The final type is `ParticleType.FIREWORK`. A firework acts like a ballistic particle in that, after it moves, its `dy` increases by one. However, there is one difference between a firework and a ballistic projectile. When a firework's lifetime decreases below zero, it *explodes*. The firework particle is removed (as usual – it now has a negative lifetime) and is replaced by a bunch of new particles representing the explosion. Specifically, create 50 new streamer particles. Each of those particles are positioned at the firework's last position. They're then given a random `dx` and `dy` between -3 and +3 and a random lifetime in the range [2, 10] (between 2 and 10, inclusive). Each streamer is given the same color, and that color should be chosen randomly. (In other words, there will be 50 streamers of the same color, but what color that is will be chosen randomly.)

To help with generating random values, we've provided you a `RandomGenerator` type with static methods for generating random real numbers, integers, and colors.

A note on fireworks: the newly-added streamers should go, as all new particles do, at the very end of the list of particles. That way, if a firework explodes during a call to `moveParticles`, the new streamer particles will also move by one step. (In fact, that's why we had you add particles to the back of the list all the way in Milestone One!)

You probably won't need to modify much existing code here to get this working, though depending on your implementation you might find that you need to decompose out some of your logic into helper functions to keep things readable.

We have provided some basic test coverage here to ensure that ballistic and fireworks particles accelerate as they are expected to. However, we haven't added tests to check for the other requirements here. You should write at least one test case that checks for some property that was not described here.

To summarize, here's what you need to do.

Milestone Five Requirements

1. Update your code to support streamer particles, ballistic particles, and firework particles.
2. Add at least one custom test case to validate that your implementation is correct.
3. Run our provided tests, plus your new tests, to make sure all the provided tests for this milestone are passing.

Some notes on this part:

- Decomposition is your friend. If you try writing all the logic to manipulate particles in the `moveParticles` function, you are going to end up with way too much code in one place and bugs will be very, very hard to track down.
- Fireworks only explode if their lifetime ends, not if they move off-screen.

Milestone Six: Watch the Show!

You now have a 2D particle system - nicely done! We've created a number of “scenes” that use your particle system as an essential component. To see your particle system in action, run the main method from `particles.gui.ParticleGUI`. Go to the bottom of the window and use the dropdown menu to select a scene and click the button to load it. Here are the six scenes we've provided for you:

- ***Fireworks:*** A fireworks show that illustrates your firework and streamer particles.
- ***Fountain:*** A simulation of a water fountain that uses ballistic particles.
- ***Magic Wand:*** Move the mouse to wave the wand and see ballistic particles. Hold the mouse down to see streamers.
- ***Photo Exploder:*** Use streamer particles to “explode” an image, then reassemble it.
- ***Snowy Day:*** A view out the window on a snowy day, made entirely of streamers.
- ***Volcano***: Use streamers and ballistic particles to simulate a volcano erupting.

Take some time to reflect on your journey here - isn't it amazing what you can do with linked lists?

There are no deliverables here - just enjoy your creation! However, feel free to create your own scenes if you’d like. If you subclass the `Scene` type in `particles.scenes`, your new scene will be integrated into the GUI program automagically and you’ll be able to see your creation in action.