What is a particle system?

A particle system is a simple physics model that deals only with point-masses and forces. That means that—as opposed to rigid-body models—objects in particle systems occupy no volume. Their behavior is quite simple but very powerful.

A particle is an object with a location and a mass. It is acted upon by any number of forces. Its acceleration is calculated by **f = ma**, where f is the vector sum of all the forces acting on it.

Particles have three properties that you may want to manipulate:
Mass: The mass of the particle. The greater the mass, the more force is required to accelerate it.
Position: You will likely want to give your particle an initial position.
Fixed: This boolean value indicates whether a particle's position is being controlled manually or by the forces in the system. A fixed particle can be attached to a single point in space, or the mouse, or may be moved simply by setting its position. Often a whole system of particles will be attached to a single fixed particle and will move with it.

A particle system takes small time-steps and updates each particle's location and velocity based on the forces acting on it.

The most commonly used forces are gravity, drag, springs, and magnets.

Gravity applies a constant additive acceleration to every particle. It can point in any direction and can be turned off.

Drag slows the movement of particles. It is equivalent to air-resistance.

Springs are forces that exist between two particles that pull the two particles together with a force proportional to the distance of their separation. Networks of particles connected with multiple spring exhibit surprising and lovely behavior.

Springs have three parameters:
Rest length: The rest length of a spring is the length at which it no longer pulls.
Strength: The strength of the spring is how hard it pulls when it is stretched.
Damping: The amount of energy absorbed by the spring as it bounces. Damping is necessary so that the spring doesn't keep oscillating for too long.

Magnets are forces that attract and repel all particles inversely proportionally to their proximity to the source. Magnets are attached to a particle so that they have a location associated with them and so that they can be conveniently drawn. But the host particle is not affected by its own magnetism. Magnets have one important parameter, their **strength**.

Using the particle systems in Processing

Particle systems are available with the PSystem plugin to Processing. For general instructions on loading plugins, see plugin system documentation. Generally, though, loading the plugin looks like this:

```
//ps is a global variable of type PSystem
PSystem ps;

...

//Then in setup()
ps = (PSystem)loadPlugin("PSystem");
```

Then you must go to "Sketch -> Add file..." and locate and add "PSystem.jar."

You use it by simply adding particles and forces (primarily springs). The rest will be taken care of by the system. Particles and springs have certain native properties and behaviors which you can change at will.

From here down, we will assume that the particle system has been loaded into a global variable called "ps."

Creating a new particle type

It is not absolutely necessary to create a new particle type because the Particle class defines a simple particle that renders as a small box. But if you want anything more exotic, you will want to write your own.

A new particle type is just a new class that extends the Particle class that gets loaded with the PSystem plugin. All particles automatically have a few properties defined on them. Their position, for instance is updated automatically by the system, and is accessed by the member variable "pos[]" which is an array of three floats. pos[0], pos[1], and pos[2] are the X, Y, and Z positions respectively. Remember that inside the particle's code itself, this variable is just called pos, but outside a particle stored in a variable called particleName, the X position would be accessed this way: particleName.pos[0]

The syntax for defining a particle type looks like this:

```
class BoxParticle extends Particle {

  BoxParticle(float x, float y, float z) {
    super(x, y, z);
    // initialization code here.
  }

  void draw() {
    push();
    translate(pos[0], pos[1], pos[2]);
    box(5, 5, 5);
    pop();
  }
}
```

This defines a simple particle that moves to its position in loop and draws a small box there. (Defining this particle is actually redundant because this is the default behavior of the "Particle" class anyway.) This techniqe of push-to-position, draw, and pop is a good model to keep. This way all particles exist in a shared coordinate system, but can draw themselves relative to their own position.

To spawn a new particle of this type, you would use the line:

```
boxPart = new BoxParticle();
ps.addParticle(boxPart);
```

You must obviously do this after you load the PSystem, but don't do it in loop() unless you want to be spawning a new particle every frame (which would quickly bog down the system).

Since particles don't do much other than fall due to gravity without springs or other forces attached, it is likely we want to hold onto a reference to the new particle when we spawn it so we can use it later. Here is an example of two particles being spawned and referenced by variables so that they can be attached by a spring.

```
BoxParticle a = new BoxParticle(0, 0, 0);
ps.addParticle(a);
BoxParticle b = new BoxParticle(100, 100, 100);
ps.addParticle(a);
Spring s = new Spring(a, b);
ps.addForce(s);
```

In the above code, we have spawned two particles and stored them in the variables "a" and "b". We have used a constructor that also sets the position of the new particle to the values entered. Then we have connected a and b with a spring, which we have assigned to a variable called "s" in case we should decide that we want to modify this spring later. We must add the particles and the force to the particle system with the "add..." methods if we want the PSystem to keep track of them.

Collisions

A simple collision model based on spherical boundaries is built in. It acts as a strong repelling magnet that is only activated if another particle's radius of collision intersects this particle's radius of collision. These are not like true elastic or inelastic collisions, but it models them well enough to look OK. Collision is turned off by default. In order for two particles to react in collision, collision must be turned on for both particles. This is done using

```
particle.enableCollision(collisionRadius);
```

where collisionRadius is the spherical radius of the particle's collision boundary. Note that this radius may or may not be directly related to how the particle is drawn. You may institute a collision boundary outside the drawn bounds of the particle to act as a force-field.

Collision can also be disabled again with

```
particle.disableCollision();
```

Collision is modelled as a strong repelling force that only activates below the collision radius. It has its minimum strength where the intrusion is minor, and takes its maximum strength when the collision is more severe. You can set the minimum and maximum collision strengths per particle:

```
particle.minCollisionForce = SOMETHING;
particle.maxCollisionForce = SOMETHING_BIGGER;
```

Old age and death

Old age and death are nothing to fear, they are well-supported. Every particle has an integer "age" field that is updated automatically each frame after it is added to the particle system.

```
particle.age
```

You can make a particle die and disappear by calling its "die()" function:

```
particle.die();
```

Die should take care of removing the particle from the particle system and removing any forces that are dependent on it.

Various stuff

Gravity:

```
ps.setGravity(Gy);
```

The setGravity function takes a float value indicating how strong gravity is in the downward Y direction. Calling this function will set gravity in the X and Z directions to zero (which is normal). It makes sense to call gravity once in your setup function because you typically only need to set the global gravity once (in setup(), say) unless there is a circumstance in your program that makes gravity change. The default gravity value is 0.03.

```
ps.setGravity(Gx, Gy, Gz);
```

This function is just like the one above, but it lets you also specify the gravity in the X and Z directions in case things in your world should be falling to the side or forwards or backwards. The default gravity values are Gx: 0.0, Gy: 0.03, Gz: 0.0.

`ps.drag = SOME_FLOAT;`
Drag is a float value indicating how much drag resists the movement of particles. It is equivalent to air-resistance. Setting the drag high is like putting it in water. Too little drag will keep the system from finding equilibrium. The default drag value is 0.03.

`particle.setPos(X, Y, Z);`
This will set "particle's" position to (X, Y, Z).

`particle.fixed()`
For a variable of a particle type called "particle" this boolean function indicates whether this particle is fixed or free to move by the forces applied to it. Note that a fixed particle can be moved around, but the PSystem itself will not be doing it automatically.

`particle.fix()`
Sets a particle to fixed.

`particle.unfix()`
Sets a particle to un-fixed.

`particle.fix(int axis, float value)`
Fixes a particle in a particular axis only. Axes are 0=X, 1=Y, 2=Z. This will in effect constrain a particle to a line. You can constrain two different axes by calling this with two different axes. A particle's "fixed()" function will not return true unless it is fixed in all dimensions.

`particle.pos`
Pos is an array of three floats that contains the particle's position in space.

`particle.mass = SOME_FLOAT;`
For a variable of a particle type called "particle" this variable is the mass of the particle. The default mass is 1.0.

`particle.enableCollision(collisionRadius);`
Enables collision with a radius of collisionRadius for this particle. Collision will only occur between two particles who both have collision enabled.

`particle.disableCollision();`
Disables collision for this particle.

`ps.addSpring(particleA, particleB);`
This is a convenience function that adds a new spring to the system between particleA and particleB. By default the spring has a rest length of 18.0, a strength of 0.005, and a damping of 0.001. If you want to modify this spring's properties, you might want to hold the spring in a variable by writing a line like this:

```
Spring s = addSpring(particleA, particleB);
```

`addSpring` is equivalent to creating a new Spring and also adding it to the PSystem.

`addSpring(particleA, particleB, strength, damping, length);`

This function is just like the above addSpring(), but you can manually specify the strength, damping, and length. If you make a spring too strong or give it too little or negative damping, you will witness a very interesting numerical explosion. It's worth trying to see it happen.

```
ps.drawForces = BOOLEAN_VALUE;
```
This field tells a particle system whether or not to draw each force. By default Springs are the only forces that know how to draw themselves, but you can add a draw method to any custom force, and it will automatically draw if ps.drawForces is true, which is the default.

```
ps.drawSprings();
```
This function will be called automatically if drawForces is true (the default), but if you set drawForces to false, you may decide to call this manually.

```
spring.restLength = SOME_FLOAT;
```
For a Spring variable "spring" this variable is the rest length of the spring.

```
spring.damping = SOME_FLOAT;
```
For a Spring variable "spring" this variable is the damping of the spring.

```
spring.strength = SOME_FLOAT;
```
For a Spring variable "spring" this variable is the strength of the spring.

```
ps.defaultSpringStrength = SOME_FLOAT;
```
This sets the default spring strength for new springs that are created after the function is called to this value. This way you can create many springs with the same strength.

```
ps.defaultSpringDamping = SOME_FLOAT;
```
This sets the default spring damping for new springs that are created after the function is called to this value. This way you can create many springs with the same damping.

```
ps.defaultSpringRestLength = SOME_FLOAT;
```
This sets the default spring rest length for new springs that are created after the function is called to this value. This way you can create many springs with the same rest length.

```
ps.addMagnet(particleA);
```
This function adds a new magnet to the system whose location is attached to particleA. By default the magnet has a strength of 1.0. If you want to modify this magnet's properties, you might want to hold the magnet in a variable by writing a line like this:
```
Magnet m = addMagnet(particleA);
```
It is possible to attach a magnet to a particle that is fixed or moving.

```
addMagnet(particleA, strength);
```
This function is just like the above addMagnet(), but you can manually specify the strength.

```
magnet.strength = SOME_FLOAT;
```

For a Magnet variable "magnet" this variable is the strength of the magnet.