

Galaxy

<i>Assignment Out:</i>	Jun 22
<i>Initial Implementation:</i>	Jun 29
<i>Final Design Due By:</i>	Jul 01
<i>Code Due:</i>	Jul 06 11:59pm

1 Purpose

This assignment is designed to introduce you to a medium scale Java system, to test your ability to design and implement Java code, to test your ability to design object-oriented programs, to give you experience with appropriate Java programming and debugging tools, and to make a program that is slightly cool and possibly even useful.

2 Overview

Galaxy is a program that simulates the evolution of a galaxy or solar system by seeing how an initial configuration of moving objects interacts via gravity and collisions over time.

We start with some number of objects each of which has an initial position, velocity, and mass. These objects interact with one another through gravity. Determining where all of them will be in t seconds is known as the **N-Body Problem**. It's very challenging to calculate exactly how the objects should move, and it would take more computer resources than we have to do it, so you're going to be approximating their trajectories as best you can. Given this, slightly different approximations could result in very different long-term results, so there is no *right* answer. Your simulation should be reasonably accurate - given a star with the sun's mass and a planet in earth's position, it should take around one simulated year to orbit the sun.

In addition to just moving around space, planets sometimes collide. Your simulation will have to handle this. However, it'll be done very simply - if two bodies collide, they turn into one object which has their combined mass, location, and weighted velocity.

3 Simulating the Solar System

The following is an overview of how to simulate the solar system. The physics behind it, especially the equations you have to use, is very important, so it has been placed in the Physics Appendix (Appendix A). Read through these steps first, then go to the Appendix to understand the equations that you'll need to implement the project.

The simulation works by calculating how all of the objects should move one step at a time by assuming that, for the duration of that step, the objects will move in a straight line. If your time step is small, like one second, then you will have a really accurate simulation. However, it would take forever to simulate even one day. If your time step is too large, like one year, then you'll get a wildly inaccurate simulation, since you'll assume that the earth would move in a straight line for one year, which isn't anything close to reality. A good time step to use is around 20 minutes, but you can change this to suit your needs.

For each time step:

1. For each object, calculate the forces that all the other objects in the simulation exert on it, and add all of these forces together. Each object interacts with each other object once, through the force of gravity. See section A.1 of the appendix for details.
2. For each object, change its velocity and position based on the force being applied to it and the length of the time step. See section A.2 of the appendix for details.
3. Once all the objects positions have been updated, check if there are any collisions. If there are, each two colliding objects should combine to form one object. See section A.3 of the appendix for details.

Then, simply repeat!

4 An Initial Implementation

Eventually, your program has to handle a large number of objects (e.g. 10,000 to 1,000,000). However, debugging the gravity computations and getting comfortable with Java and all the programming tools is going to be very complex if you try to do everything at once. Therefore, we recommend that you start of by writing the code to handle a relatively small number of objects with a understandable configuration first. Once you get this working, you can start worrying about the data structures and changes that will be needed to get your program to scale to much larger numbers of objects.

We suggest that you start with a simple solar system consisting of a star (massive central object) and 100 smaller objects orbiting the start.

The system should be different each time you run your program, given the following constraints (note that this will make more sense once you understand the physics behind the project):

- The star is at $(0, 0, 0)$, and has the mass and radius of the sun. The mass is 1.989×10^{30} kg, and the radius is 695,000,000 m.
- The masses of all 100 objects should be approximately $1.18 \times 10^{25} \pm 1\%$ kg. The easiest way is to give them all the same mass, but they can vary if you like.
- Each object should initially be in a stable orbit around your star with a radius greater than Mercury's orbit and less than Mars'. That means the distance between the star and each object should be between 57,910,000,000 m and 227,940,000,000 m. See A.4 for how to create a stable orbit.

- Space *is* three-dimensional. It is acceptable for all your z coordinates to be zero, making your solar system, in effect, a flat disk. However, if you want a slightly more exciting simulation, you can add a random component to the z axis - you can vary it from around -60,000,000,000 meters to +60,000,000,000 meters. This would also be a good way to make sure that z coordinates are taken into account in your calculations.

Note that you might want to try your program on simpler cases before doing the full simulation. For example, a single object, or a pair of objects opposite each other, should yield a stable solution; setting an initial velocity of 0 for an object will test its gravitational attraction to the sun (it should fall straight in). It's also probably a good idea to use the same initial configuration when debugging rather than a different one each time you run.

5 The Initial Design

We'd like you to write up your initial design to us much like you would at a medium-sized software company when the idea is in its initial small public presentation stage (e.g. presenting to your "team" of 3-6 people). This is less formal than a full description of exactly how the code will work, but it still contains a complete picture of the project which another person can easily understand. To be specific, we'd like to see:

- A complete class diagram, showing all classes and interfaces, showing run-time relationships (e.g. class A creates class B, and class C has a collection of class A instances, etc.)
- Inheritance and "implements" relationships among the classes and interfaces
- Method names and signatures (e.g. `void foo(Atype a, Btype b, int bar)`) with a short description for each (a sentence at most)
- Each class and interface should have a few sentences explaining its purpose and responsibilities
- Consistent, easily comprehensible symbols, which need not be perfect UML, or even like UML, but one should be able to understand what they mean without any additional explanation (use a legend if you think it would help). UML will come later.
- Descriptions of test cases you'd like to use to test various aspects of your code as you go and at the end. Be fairly specific and be sure to include the expected result with each test case.

There are also a lot of things we *don't* want to see. These include:

- A poor diagram, which you hope to make up for with a better verbal description
- Any filler, fluff, or verbosity that doesn't improve the diagram. If it takes four words to explain something really simple, only spend four. Save your words for the things which actually have complexity, but then spend them generously. We are not grading based on word count.

- Any other materials or additions. Your diagram should be awesome enough to stand on its own.
- Differences between your diagram and your “current design”. Your diagram should be legible and up-to-date when you present it.

We really want you to get *a lot* out of this design check.

6 Using the SolarDraw GUI

The goal of the project is to convince the TAs, as well as yourself, that your code works. Probably the easiest and most effective way to do this is by being able to physically see your solar system. To help with this, we’ve provided a GUI program, called the SolarDraw GUI, which will render your solar system for you using OpenGL.

6.1 Using SolarDraw in your code

Javadocs are available from the website for detailed info on all the functions SolarDraw supports, but here’s all you should need to know to use it in your code.

First, to use the GUI, you must import this package:

```
import edu.brown.cs.cs032.solardraw.*;
```

You should make your top level class implement the `SolarDraw.Control` interface. This means you need to implement the following methods:

```
interface SolarDraw.Control {
    double getTime();
}
```

For now, just define an empty function. `getTime()` should return the current time of your simulation in seconds. `getUPS()` is optional, see the next section for details.

Now, in your top-level object, create a `SolarDraw` instance like so:

```
SolarDraw solarDraw = SolarDraw.Factory.createSolarDraw(this);
```

You have to tell `SolarDraw` about all the objects that you want it to draw. You do this by registering them:

```
solarDraw.registerObject(myObject);
```

`myObject` is an object which has to implement the `SolarDraw.Object` interface, which is pretty self-explanatory:

```

interface SolarDraw.Object {
    double getMass();
    double getRadius();
    double getX();
    double getY();
    double getZ();
}

```

The GUI will constantly call these functions whenever it re-draws them. Once you've registered an object, you don't have to worry about it anymore.

When you're done adding your objects, you want to start rendering your solar system. To make the GUI show up, call:

```
solarDraw.begin();
```

You should only call this method once!

To unregister an object (in other words, remove it), simply set its mass to zero, and the GUI will automatically unregister it. You will have to do this when two objects collide.

IMPORTANT NOTE: The SolarDraw GUI currently has a bug. If you've only registered one object with it, it's not going to display it; you're going to see a blank screen. When testing to see that the SolarDraw GUI works, make sure that you add at least two objects. This is due to a problem with its automatic zooming.

6.2 The Interface

The GUI has a few features which are useful. These are the controls that it supports:

Page Up	Zoom in
Page Down	Zoom out
Arrow Keys	Rotate by 30 degrees
Home	Reset the GUI
Hold Right Mouse Button	Rotate the screen
Shift + Hold Right Mouse Button	Pan the screen

A few notes:

- There is no key or button to close the SolarDraw window. Just use your window manager's standard close button.
- The SolarDraw GUI likes to recenter its view, sometimes in a sudden way, so don't be alarmed if things are a little jerky.
- The GUI doesn't draw things necessarily to scale. If you zoom in and see your planets colliding, but your collision code isn't verifying that, they might actually be far apart enough to not collide, but the GUI is exaggerating the size of the spheres to make them easier to see. So, when checking collision detection code, don't rely on the GUI, but use debugging print statements instead.

If you have any problems with the GUI, please report them to `cs032tas@brown.edu`. If you have any suggestions as to how to make the interface more informative or more helpful, please tell us – we are looking for suggestions.

As we mentioned before, we won't be grading for speed. However, there are a few important tips that won't take much effort to implement and can get your Solar to run at up to 1000 UPS for 100 objects.

Make sure to tell us whichever one of these you've done in your `README`. If you've done any other optimizations which you think improved your project's speed a lot, be sure to mention those as well.

7 Java Math

We've said that speed doesn't matter. However, there is an important thing to know when doing your project - **do not use the `Math.pow` function to square numbers**. In other words, if you have a `double x`, the following code:

```
Math.pow(x, 2.0);
```

is about **five to six times slower** than doing

```
x * x
```

Whenever you square a number, such as in distance calculations, make sure you don't use `Math.pow`. Since this is done several times in the force calculation, which gets executed thousands of times a second, this will be a gigantic speed upgrade.

The reasoning behind this is that `Math.pow` is a very general function which can handle any exponents, including fractional and negative ones. It doesn't check the special cases of squaring or cubing a number, so it'll take the same amount of time to do that as to raise a number to the 3.43284th power, which is a very inefficient calculation.

8 Demo

To run the demo of the simple implementation, use this command:

```
/course/cs032/demos/solar [planets]
```

The demo takes a single argument, the number of objects that you want in the solar system. If you do not specify the number of objects you want, the demo will default to 100.

9 Getting Started

We encourage you to use Eclipse to develop this project. Here are the steps you need to take to get all set up:

1. Copy the stencil code to your home directory:

```
cs032_install solar (or /course/cs032/bin/cs032_install if cs032 is not in your path)
```

2. Run eclipse: `eclipse` . If it says file not found, type in the full path: `/contrib/bin/eclipse`
3. Create a workspace if you haven't already.
4. Go to File → New Project. Choose Java Project. Give it a name.
5. In the resulting dialog box, select “Create Project from Existing Source”. Navigate to the solar folder which you created in Step 1. Click Next.

6. Now we have to add libraries to get the SolarDraw GUI to work. Go to the “Libraries” tab. Click “Add External Jar.” In the file selection dialog box, choose:

```
/course/cs032/lib/solardraw.jar .
```

7. Add another external jar: `/pro/java/linux/software/jogl/jogl-1_0_0/lib/jogl.jar` .
8. Click Finish. The project will now be created and will show up on the left.
9. Right-click on this project. Go to Run As → Run Configurations. Go to the “Arguments” tab. Under “VM Arguments,” enter:

```
-Djava.library.path=/pro/java/Linux/software/jogl/jogl-1_0_0/lib
```

10. Click Close. The project should now be ready to run. You can run it by right clicking on `SolarMain.java` located by in the `javasrc - > edu.brown.cs032.solar` package.

10 Moving on to a full Galaxy

10.1 New Force Calculation Algorithm

Simulating 1,000+ objects requires a different algorithm than simulating 100. Try running your `Solar` project on a huge number of masses to see for yourself.

You're free to use any approach you like to allow your `Galaxy` simulator to handle 10,000 objects, whether you invent it yourself or find it through research. However, we strongly recommend that you use the **Barnes-Hut Algorithm**.

The intuition behind it is that to find the effect of a group of distant objects on a certain planet, we can treat the group of planets as a single planet without losing too much accuracy.

The algorithm actually consists of two components – an octtree implementation as well as an algorithm that is applied to the octtree. Rather than try to explain the algorithm here, we've decided to give you links to websites that do just that. These can be hard to read and understand, so for that reason we will answer questions in detail on this in class. It's much easier to explain in person, and with examples, so if you have trouble understanding the websites, be sure to come to the help session!

That being said, check out:

http://www.flipcode.com/archives/Introduction_To_Octrees.shtml

for an explanation of octrees. Note that, since your planets are constantly moving, you're going to have to recalculate your octree every frame. You could try recalculating it only every few (2 to 4) frames for some extra speed, since if your time step is small enough it should still be reasonably accurate, but this isn't as accurate and is not required.

<http://www.cs.berkeley.edu/~demmel/cs267/lecture26/lecture26.html> provides a good explanation of the Barnes-Hut algorithm itself, as well as a small section on octrees. Note that these links are available on the course website as well, in the Algorithms / Docs section.

If after the websites and the help session you still have trouble with the algorithm, come to TA hours and we'll be glad to help you understand it.

10.2 New Collision Detection Algorithm

Note: This section makes more sense once you understand the octree algorithm.

Now that you have an octree set up for the new force calculation, you can also make collision detection a lot more efficient. The reasoning behind collision detection is as follows: if a particular node in the octree does not intersect with a planet, then nothing in that node can possibly intersect with it either, so you can ignore the entire node.

So, intersect every planet in your solar system with the root octree node. The collision between a planet p and a node is done recursively as follows:

- If the node is a leaf node (has no children octree nodes), and it has no planets in it, then there's no collision.
- If the node is a leaf node and has a planet, then check collision between p and that planet in the same way as in Solar.
- If the node isn't a leaf node then, for every child node:
 - See if p intersects the child node's bounding box. You can do this by adding the planet's radius to every dimension of the child node's bounding box and then seeing if that box contains the planet's center.
 - If p intersects the child node, then recursively collide p with that child node. If any collisions happened, then return true. Otherwise, check the other child nodes.
 - If p doesn't intersect the child node, then just continue to the other child nodes.

10.3 XML

Your project should be able to read and write the state of the simulation to a file using our specified XML format. You're welcome to share your test input files (and their resulting output) with other CS32 students to test your code.

The specification for the XML we want you to read and write consists of only four tags.

SOLAR the root tag for the document (i.e. the outermost tag, which holds all the content inside). It has one required attribute:

TIME The number of simulated seconds the simulation has run at the moment the XML file is saved. When you first create a galaxy, start with its `time` attribute as 0. If you save its XML later after simulating it, update the time.

OBJECT the only tag allowed directly within a `solar` tag. Every object in the system is described by an `object` tag. Its attributes are:

NAME some name you give the object

MASS the object's mass (in kilograms)

RADIUS its radius (in meters)

Within the `object` tag there must be exactly two other tags, `position` and `velocity`.

POSITION always found nested in an `object` tag, it specifies an object's position, with the following attributes:

X object position's *X*-coordinate (in meters)

Y object position's *Y*-coordinate (in meters)

Z object position's *Z*-coordinate (in meters)

VELOCITY always found nested in an `object` tag, it specifies an object's velocity, with the following attributes:

X object velocity's *X*-coordinate (in meters/second)

Y object velocity's *Y*-coordinate (in meters/second)

Z object velocity's *Z*-coordinate (in meters/second)

Here's a sample XML document your program should be able to handle.

```
<SOLAR TIME='0.0'>
  <OBJECT NAME='Object_0' MASS='6.93802901065139E23' RADIUS='1271511.650378473'>
    <POSITION X='-2.966473042307841E9' Y='7.289090278627592E8' Z='772.2514875' />
    <VELOCITY X='-161.88780633549223' Y='-658.841906212116' Z='0.0' />
  </OBJECT>
  <OBJECT NAME='Object_1' MASS='5.7352863630113265E23' RADIUS='1193328.48932346'>
    <POSITION X='-1.8524168835642315E10' Y='4.746119148457207E9' Z='33685.441' />
    <VELOCITY X='-97.34967716226407' Y='-379.95713959052785' Z='0.0' />
  </OBJECT>
</SOLAR>
```

For everything you need to know to do XML Saving and Loading, take a look at appendix B.

10.4 Adaptive Time-Stepping

Having a fixed time step leads to many problems. The time step could be fine for one solar system, but way too small for a very expanded galaxy, and way too large if you have a very small solar system. Furthermore, objects will tend to go through each other if they get very close, and have inaccurate orbits.

The choice of the proper time step is dependent on a number of factors such as the speed of the objects and their distance from one another. For every step of the simulation, the following criteria should be met:

- To make sure the simulation is accurate, no planet should move more than 1/100th of the average distance between two objects in one gravity computation.
- To make sure that collisions are detected, no planet should move more than half of the way through any other planet.

You have to calculate a time step every frame that makes sure every planet satisfies these criteria. To do this you must limit your time step in two ways.

- You must keep track of the average distance between all objects as well as the maximum velocity of any object. You then must make sure that the fastest object will not travel more than 1/100 of this distance. So,

$$t_{max} = d_{avg}/100/v_{max}$$

The timestep cannot exceed t_{max} , or your simulation will be too inaccurate.

- For every pair of objects, you have to find out how much time it would take for them to go halfway through each other. An object going at velocity \vec{v} travels $\vec{v}t$ meters per second. So, for every pair of planets you look at, see how long it would take them to travel $d - \frac{r_1+r_2}{2}$ meters, where d is the distance between them and r_1 and r_2 are their radii. You can either:

- Find the magnitude $|\vec{v}|$ of each of the planets velocities:

$$|\vec{v}| = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

add them together, and divide the distance between them by this amount.

$$t_{max} = \frac{d - \frac{r_1+r_2}{2}}{|\vec{v}_1| + |\vec{v}_2|}$$

- The previous method isn't too accurate, since the two planets are not necessarily always moving directly towards each other. You will always err on the side of caution, so it is acceptable, but to be more efficient you'll have to calculate the **projection** of each planet's velocity onto the vector representing the distance between them. This isn't necessary, but if you're up for it:

First, find the distance vector between them:

$$\begin{aligned}\vec{d} &= \vec{p}_2 - \vec{p}_1 \\ (d_x, d_y, d_z) &= (p_{2x} - p_{1x}, p_{2y} - p_{1y}, p_{2z} - p_{1z})\end{aligned}$$

where p is a planet's position.

Then, find the **scalar projection** s of each planet's velocity onto the distance vector:

$$s = \frac{\vec{d} \cdot \vec{v}}{|\vec{d}|}$$

where

$$\vec{d} \cdot \vec{v} = d_x v_x + d_y v_y + d_z v_z$$

This will give you exactly how much each object will move in the direction given by \vec{d} . It will be the same as the previous method if the object is moving in that direction, smaller if it's in a different direction, and negative if it's in the opposite direction.

Compute s_1 and s_2 and use them in place of $|\vec{v}_1|$ and $|\vec{v}_2|$ in the previous method to calculate t_{max} .

Note: You will have to use $\vec{p}_1 - \vec{p}_2$ for \vec{d} when calculating s_2 , because you want to see how fast the second planet is moving towards the first, so you want your direction to point accordingly. You don't have to completely recalculate \vec{d} though – you can just negate the one you used before, or, more simply, use the same \vec{d} but negate s_2 .

t_{max} now represents the maximum timestep in your simulation to ensure collision between this pair of planets.

You will end up with many different values for t_{max} : one to ensure accuracy, and one for each pair of planets to ensure collisions. Pick the smallest of these in order to satisfy all the criteria.

Note: Since you'll implement an octtree, you won't actually be iterating between all pairs of objects anymore. Doing so just for the sake of picking a time-step will slow the simulation down an unacceptable amount. Instead, you should be doing these calculations as you're doing your force calculations. If you end up doing a force calculation between an object and a leaf node, then you can do these calculations between the object and the object in the leaf node directly. If, however, your program decides that a node is far enough away to be considered as a single object, then:

- For the average distance calculation, use the node's center of gravity to calculate the distance, and weigh it by the number of objects in that node.
- For the collision ensuring calculation, if you skip an entire node, then your object is too far to be at risk with colliding with anything in that node, so you don't have to limit the time-step at all.

The second part of these calculations seems a bit inefficient – you can have 1000 objects all millions of kilometers from each other, but then you have another two which are just a few thousand kilometers away. This will result in the entire simulation slowing down for just those two objects. There is a way around this.

1. Pick a timestep based only on the average distance calculation.
2. If two objects are too close to each other, and are in danger of missing a collision, then do multiple force calculations for just those two objects. For example, you might split the timestep into two. You would first do a force calculation, update the two object's new positions and velocities, then do another force calculation using their new positions and velocities and update them again. If they are close enough together, this will be considerably more accurate. Note that you should probably do collision detection after each interval.

10.5 A few more things

- **Galaxy** is 3-dimensional. When you set up your scenes, make sure you spread things out in the z -dimension as well. Feel free to be creative: try simulating two galaxies colliding, a galaxy with a huge black-hole (i.e. a massive object) in the center, or perhaps two black-holes orbiting each other in the center of a galaxy. Of course, when you test initially, it's a good idea to set up some simple situations whose behavior you can predict by hand. Also make sure you have a test case which clearly demonstrates your adaptive time stepping.
- If an object 'escapes the system' (that is, gets very far away from everything else, say 10 times farther away than the initial maximum distance between two objects), you're allowed to delete it and forget about it.
- Your program should take command line arguments. It should at least be able to take the name of the XML file containing the initial configuration. You can add other arguments if you want (e.g. parameters that control your simulation and thus allow you to experiment without recompiling).

11 Design Check

Once you have an understanding of the full problem, you should design your solution. Be prepared to give a presentation and handin similar to what we done with the initial design for your full design.

Please have a testing plan as part of your hand in here. Unlike the initial design check, we expect you to come with an electronically generated UML diagram. We recommend using Argo, which you can launch by using the following script:

- `/course/cs032/pub/cs032argo`

We also expect to see a write-up of test cases similar to what you did for checkpoint 1 in lab01. We want to see JUNIT test cases (pseudocode please) for all the functions that generate output,

including any supporting math or physics functions that you plan to write. Also, include in your test plan test cases that test the visual output of your program.

This is worth 10% of your grade.

12 Demo

The demo can be run in one of two ways:

- `/course/cs032/demos/galaxy -random <numplanets> [optional timestep]`
This will create a galaxy with a given number of randomly placed planets. If an optional timestep is specified, it will run using that timestep instead of doing adaptive timestepping.
- `/course/cs032/demos/galaxy <xmlfile> [optional timestep]`
This will load the given xml file. `optional timestep` works the same way.

13 Handin

Your grade will be based on your design check and your handin. Your handin should be very easy to compile and run. Your code should be easy to read.

Make a `README` file which explains any quirks in your code, as well as any special instructions about running your program. You should also mention any design decisions you made which you found troubling.

Handin with the following command, which you should run from the directory which contains `build.xml` so we'll get everything.

```
/course/cs032/bin/cs032_handin galaxy
```

A Physics Appendix

This contains all the equations you'll need to get your project running. **Important note:** All of this math will be done in kilograms, meters, seconds, and Newtons. The equations will only work with these units, so make sure that you are consistent in your code with how you represent them.

A.1 Calculating Forces

A force is made up of the magnitude of the force, or how powerful it is, as well as the direction the force is acting in. For the force of gravity, the magnitude is determined by this formula:

$$F = \frac{Gm_1m_2}{d^2}$$

where m_1 is the mass of the first object, m_2 the mass of the second, d the distance between the two, and G the **universal gravitational constant**, which is

$$G = 6.674 \times 10^{-11} \frac{Nm^2}{kg^2}$$

The distance d is determined in a similar way to that in two-dimensional space, except you have to account for the z coordinates as well:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

Now you have a magnitude. We need a way to represent the direction of our force. However, rather than thinking of the force as one magnitude and one direction, it's much easier to think of a force as three separate forces - one acting only on the x-axis, one on the y, and one on the z. If we split our force in this way, then combining forces becomes very easy - you simply add the components together separately:

$$\begin{aligned} F_1 &= (F_{x_1}, F_{y_1}, F_{z_1}) \\ F_2 &= (F_{x_2}, F_{y_2}, F_{z_2}) \\ F_1 + F_2 &= (F_{x_1} + F_{x_2}, F_{y_1} + F_{y_2}, F_{z_1} + F_{z_2}) \end{aligned}$$

This is vital, since you'll be summing hundreds of forces together.

What we need is a way to break apart a force into its three components, one on each axis. This is done using the following formulas:

$$F_x = F \frac{x_2 - x_1}{d}, F_y = F \frac{y_2 - y_1}{d}, F_z = F \frac{z_2 - z_1}{d}$$

Make sure you keep the order of the subtraction straight. If you accidentally subtract the second object from the first, then they will repel each other, and you'll get a very strange-acting solar system.

So, to calculate all the forces acting on an object at a given time, you:

1. Calculate the force that each object is exerting on it separately.
2. Split this force into its three components.
3. Add all of the forces together by summing their components separately.

To see how to affect your object based on the force acting on it, see the next section.

A.2 Approximating the Motion

For each object, you now have three forces acting on it, one for each axis. Using these forces, we can update each dimension separately.

A force acts on an object over a period of time. The length of this time period is the time step that you've decided to use for your simulation. Make sure that if, for example, you decide to use a time step of 20 minutes, you plug 1200 into all these equations, since they all assume you're using standard units, which, for times, are seconds.

First, we have to determine the acceleration of the object over this time. You probably remember the equation

$$F = ma$$

where F is the force along one of your axes calculated from the previous section, and m is the mass of the object. We just re-write this to solve for a :

$$a = \frac{F}{m}$$

This is where the approximation part comes in: a should actually be a continuous function during the time period, but we're going to pretend it's constant, which makes everything a lot simpler.

The simplest method to find the new position and velocity is called **Euler Integration**. Using this method, the new velocity is

$$v = v_0 + at$$

and the new position is

$$p = p_0 + v_0t + \frac{1}{2}at^2$$

where v_0 and p_0 are the object's old velocity and position and t is the length of the time step.

You have to perform these calculations three times - one for each of the forces acting on your object. Once you have the new position and velocity, simply update the object with them, and you're done!

Note: Make sure that you don't move an object until all of the forces on all of the objects have been calculated. All these forces are supposed to be applied *simultaneously*, and if you move an object prematurely all the force calculations involving it are going to be off.

If you feel that you want to try a more accurate motion approximation for some extra credit, see section B.2 for some info on methods you can look up.

A.3 Collisions

To make things simpler, all of your objects are going to be spheres. So, in addition to a position, your objects have a radius. When running your simulation, you have to check whether two objects are actually touching each other. This is a simple check; two objects are colliding when:

$$d < r_1 + r_2$$

where d is the distance between them and r_i is the radius of each object. This checks if they are touching exactly. However, the simulation is just an approximation, and if objects are moving very quickly and your time step is too large, they can actually skip over each other when they should be colliding. To attempt to account for this, you should scale the right side by a factor k :

$$d < k(r_1 + r_2)$$

This effectively treats the spheres as if they were bigger, which increases the chance of them colliding. Make sure that k is not too large, though. If it is, objects which aren't actually colliding will behave as if they are. A k somewhere between 5 and 20 seems to work well, but try different values yourself and see what works well.

When two objects collide, they should become one object. The new object should have a mass which is the sum of the masses of the original two objects:

$$m_{new} = m_1 + m_2$$

The object's new position should be the center of the collision, which is just the average of their positions:

$$x_{new} = \frac{x_1 + x_2}{2}, y_{new} = \frac{y_1 + y_2}{2}, z_{new} = \frac{z_1 + z_2}{2}$$

The object's velocity should be the **weighted average** of the two original velocities, weighted by the objects' masses. The idea is that the heavier object will influence the new object's velocity more.

$$\begin{aligned} v_{x_{new}} &= \frac{v_{x_1}m_1 + v_{x_2}m_2}{m_1 + m_2} \\ v_{y_{new}} &= \frac{v_{y_1}m_1 + v_{y_2}m_2}{m_1 + m_2} \\ v_{z_{new}} &= \frac{v_{z_1}m_1 + v_{z_2}m_2}{m_1 + m_2} \end{aligned}$$

And that's all you need to set the new object on it's way.

A.4 Establishing a Stable Orbit

Once you've selected a random initial position for an object, as well as a mass, you need to initialize its velocity so that it doesn't simply fall straight into the sun. **Note:** These calculations assume the sun is at position $(0, 0, 0)$.

First, calculate the force of attraction between the object and the star. This is done the same way as in A.1. The speed of the object for a stable orbit is going to be

$$v = \sqrt{\frac{Fd}{m}}$$

where F is the force of attraction, d is the distance between the object and the sun and m is the object's mass.

The velocity components are going to be:

$$v_x = \frac{vy}{d}, v_y = -\frac{vx}{d}, v_z = 0$$

Note that x and y are switched in the equations.

To make the simulation more interesting, you should perturb the v_x and v_y values at random by a factor up to 10% of its value. This will give you slightly elliptical orbits and encourage collisions between objects.

B XML Primer

Here's a quick overview on how to get started with XML in java. In this section, we're going to write code to load and save files in the following format:

```
<World name="Quuxo">
  <Country name="Footrania">
    <State name="Barabara" size="14" />
  </Country>
  <Country name="Bazonia">
    <State name="Quxa" size="199" />
  </Country>
</World>
```

B.1 Theory

XML is represented as a tree structure. It consists of nodes – things like `World`, `Country`, and `State`.

Each document has one root node – in this case, the `World` node. Each node has zero or more children – `World` has two children nodes, the two `Country` nodes. Each `Country` node has one `State` node as a child, and each `State` node has no children.

Each node has one or more attributes – `World` nodes, in this example, have a `name` attribute, while `State` nodes have both `name` and `size` attributes. Note that each attribute is just a `String` – even though `size` represents a number, the XML file doesn't know this, so your code must convert it to an `Integer` or a `Double`.

B.2 Imports

First, to make sure you have everything, import the following things in the class file that will handle your saving and loading:

```
import org.w3c.dom.*;
import org.xml.sax.*;
import java.io.*;
import java.text.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
```

B.3 Loading

Loading is done using DOM, which stands for **Document Object Model**¹ DOM parses the entire XML file at once and provides ways to access the internal tree structure.

B.3.1 Loading the File

First, you want to load your entire XML file into a `Document` variable. This is done as follows:

```
Document document;
String XMLFileName = "galaxyfile.xml";
try {
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    DocumentBuilder builder = factory.newDocumentBuilder();
    document = builder.parse( new File(XMLFileName) );
} catch (Exception e) {
    //Some error occurred -- print it out and exit
}
```

It's rather convoluted, and it isn't too important to know exactly what each of those lines does or exactly what exceptions they return, so you can just copy that straight into your loading code. The end result is that `document` contains your parsed XML file.

B.3.2 Getting the Root Node

Each node in the tree structure is represented by the `Node` class. To get the root node – in our case, the `World` node, you do:

```
Node world = document.getDocumentElement();
```

B.3.3 Getting Node Names

To get a node's name, do `Node.getNodeName()`. You probably want to make sure that each node you are accessing has the right name; if it doesn't, the XML file you're reading is an invalid format, and you should return an error.

```
if (!world.getNodeName().equals("World")); //Error - do something
```

B.3.4 Getting Node Attributes

You can get a node's attributes as a map by doing `Node.getAttributes()`. This returns a `NamedNodeMap`. To access an attribute by name, you do `NamedNodeMap.getNamedItem()`. This actually returns a `Node`, so you have to call the `Node.getNodeValue()` function to get the actual attribute as a `String`. Note that node names and attributes are case-sensitive.

¹There is an alternate way to load XML files, called SAX, but DOM is simpler to implement and is all that will be covered here.

```
NamedNodeMap attributes = world.getAttributes();
String worldName = attributes.getNamedItem("name").getNodeValue();
```

B.3.5 Getting Node Children

Now that we have the world name, we want to see what countries are in the world. We do this by getting the world node's children nodes and iterating through them:

```
NodeList countryNodes = world.getChildNodes();
for (int i = 0; i < countryNodes.getLength(); ++i)
{
    Node country = countryNodes.item(i);
    System.out.println("Country name = " + country.getNodeName());
}
```

B.3.6 Text Nodes

An important thing to note about XML – whitespace does matter. That means that there is a difference between

```
<World name="Quuxo">
  <Country name="Footrania" />
</World>
```

and

```
<World name="Quuxo"><Country name="Footrania" /></World>
```

The first one has text nodes added in. So the children in the `World` node in the first example are actually:

1. A `Text` node whose value is `"\n "`
2. A `Node` whose name is `"Country"`
3. Another `Text` node whose value is `"\n"`

It's easy to test whether a node is a text node or a real node:

- `Node.getNodeName()` will return `"#text"` if the node is a text node.
- `Node.getNodeType()` will return `Node.TEXT_NODE` if the node is a text node.

You should check for this in either way and ignore text nodes if they come up.

B.3.7 Example

Now you can do all the things with `country` that you can do with `world`, including iterating through its children. For example, if you want to get the first country's first state's name and size, you would do:

```
NodeList countryNodes = world.getChildNodes();
Node firstCountry = countryNodes.item(1); //skip the text node
NodeList stateNodes = firstCountry.getChildNodes();
Node firstState = stateNodes.item(1); //skip the text node
NamedNodeMap stateAttrs = firstState.getAttributes();

String stateName = stateAttrs.getNamedItem("name").getNodeValue();
int stateSize = Integer.parseInt(
    stateAttrs.getNamedItem("size").getNodeValue());
```

Note that `NodeList.item` returns `null` if the index is out of bounds. Also note that to get `stateSize` as an `int`, we had to call `Integer.parseInt()` on the `String` that `getNodeValue()` returned.

B.4 Saving

Saving is done by creating a `Document` element, creating a root `Node`, and adding children to that root node one by one.

B.4.1 Creating the Document

Creating the `Document` is done as follows:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document document = db.newDocument();
```

Remember to wrap all that in a `try/catch`, since they can each throw various exceptions.

B.4.2 Creating Nodes

You have to use the `document` to create any nodes in the XML file. This is done using `Document.createElement()`:

```
Element world = document.createElement("World");
```

`Element` is a subclass of `Node` which allows for attributes to be set.

B.4.3 Setting Node Attributes

This is done by using `Element.setAttribute()`:

```
world.setAttribute("name", "Quuxo");
```

where the first parameter is the attribute name and the second is the value.

B.4.4 Appending Children

Both `Documents` and `Elements` can have children appended to them. A `Document` can have only one child – this is the root node. A `Element` can have any number of children.

So, the code to create the XML file we've been using as an example would be:

```
Element country = document.createElement("Country");
country.setAttribute("name", "Footrania");
Element state = document.createElement("State");
state.setAttribute("name", "Barabara");
state.setAttribute("size", "14"); //note that 14 is a String, not a number.
country.appendChild(state);
world.appendChild(country);
/* .
   . Similar code for the other country, "Bazonia".
   .
  */
document.appendChild(world);
```

B.4.5 Saving the Document

Now that you have a `Document` representing your desired XML file, you have to actually save this to a file on disk. This is also done in a very convoluted way, much like loading an XML file, so you can just copy the following code and use it as-is:

```
String fileName = "output.xml";

TransformerFactory tFactory = TransformerFactory.newInstance();
Transformer transformer = tFactory.newTransformer();
transformer.setOutputProperty(OutputKeys.INDENT, "yes");

DOMSource source = new DOMSource(document);
StreamResult result = new StreamResult(new File(fileName));
transformer.transform(source, result);
```

This will save the document to "output.xml". This can also throw some nasty exceptions, so be sure to wrap it in a try/catch.