Project 0: N-Body Simulation

Introduction

In this assignment, you will write a program to simulate the motion of *N* objects in a plane, mutually affected by gravitational forces, and animate the results. Such methods are widely used in cosmology, semiconductors, and fluid dynamics to study complex physical systems. Ultimately, you will be creating a program **NBody.java** that draws an animation of bodies floating around in space tugging on each other with the power of gravity.

Here's an animation of a completed project running with some planets in our solar system. The animation repeats after one earth year, but your program should continue.

If you run into problems, please review the entire assignment including the FAQ section and previous posts, before making a new post to Piazza. We will keep the FAQ section updated as questions arise during the assignment.

Downloading the Starter Files

You can find the files in this repository: https://coursework.cs.duke.edu/201spring18/nbody-start

The starter code contains several image files in the **`images`** folder, the beginning of the **`NBody`** class in **`NBody.java`**, and library files for drawing and audio. You'll likely want the Eclipse configuration files (e.g., .classpath and .project) that you'll import automatically using Git.

For using git, clone the repository at this URI
`git@coursework.cs.duke.edu:201spring18/nbody-start.git`

Note: When you fork & clone the starter repository and import to Eclipse you'll see several red-X error indications. Each time you successfully complete and test in the steps below some of the red-X's will be removed.

Tasks to Complete

You will create, develop, and test the class **`Body`**[1]. We provide tests for each method you write. Then you'll complete and test the class **`NBody`** which runs the celestial, gravitational simulation. You will complete the project by writing about the simulation in your *REFLECT.txt* file. You'll complete a *REFLECT* document for all projects in Compsci 201.

The write-up below elaborates and explains these steps.

1. Create a project with the starter files
2. Create a **`Body`** class
3. Implement and test constructors and methods in the **`Body`** class
    A. Constructors
        1. Construct Body from initial values
        2. Construct Body from another Body
    B. Methods
        1. **`calcDistance`**
        2. **`calcForceExertedBy`**
        3. **`calcForceExertedByX and calcForceExertedByY`**
        4. **`calcNetForceExertedByX and calcNetForceExertedByY`**
        5. **`update`**
4. Implement and test two static methods in the **`NBody`** class
    A. **`readRadius`**
    B. **`readBodies`**
5. Implement and verify the code that runs a planetary simulation in the **`NBody`** class
6. Complete the REFLECT.txt file

Understanding the Physics

This optional section describes some of the physics behind the simulations. You may need to refer back to these explanations when implementing methods in the **`Body`** class.

[1] "Body" here refers to astronomical objects such as planets and stars, not the human body.

Gravitational Forces

Our **`Body`** objects will obey the laws of Newtonian physics. You do not need to fully understand the laws of physics used here, but you will need to understand some basic geometry and how the simulation based on these laws works. In particular, bodies in the simulation will be subject to:

- **Pairwise Force**: *Newton's* Law of Universal Gravitation asserts that the strength of the gravitational force between two particles is given by the product of their masses divided by the square of the distance between them, scaled by the gravitational constant *G* ($6.67 * 10^{-11}$ N-m^2 / kg^2). The gravitational force exerted on a particle is along the straight line between them (we are ignoring here strange effects like the curvature of space). Since we are using Cartesian coordinates to represent the position of a particle, it is convenient to break up the force into its x- and y-components (F_x, F_y). The relevant equations are shown below. We have not derived these equations, and you should just trust us.

  - $F = G\frac{m_1 m_2}{r^2}$
  - $r^2 = dx^2 + dy^2$
    (Note dx is delta/difference between x-coordinates, similarly for dy).
  - $F_x = F\frac{dx}{r}$
  - $F_y = F\frac{dy}{r}$

  Note that force is a vector (i.e., it has direction). In particular, be aware that *dx* and *dy* are signed (positive or negative). By convention, we define the positive x-direction as towards the right of the screen, and the positive y-direction as towards the top.

- **Net Force**: The *principle of superposition* states that the net force acting on a particle in the x- or y-direction is the sum of the pairwise forces acting on the particle in that direction.

In addition, all bodies have:

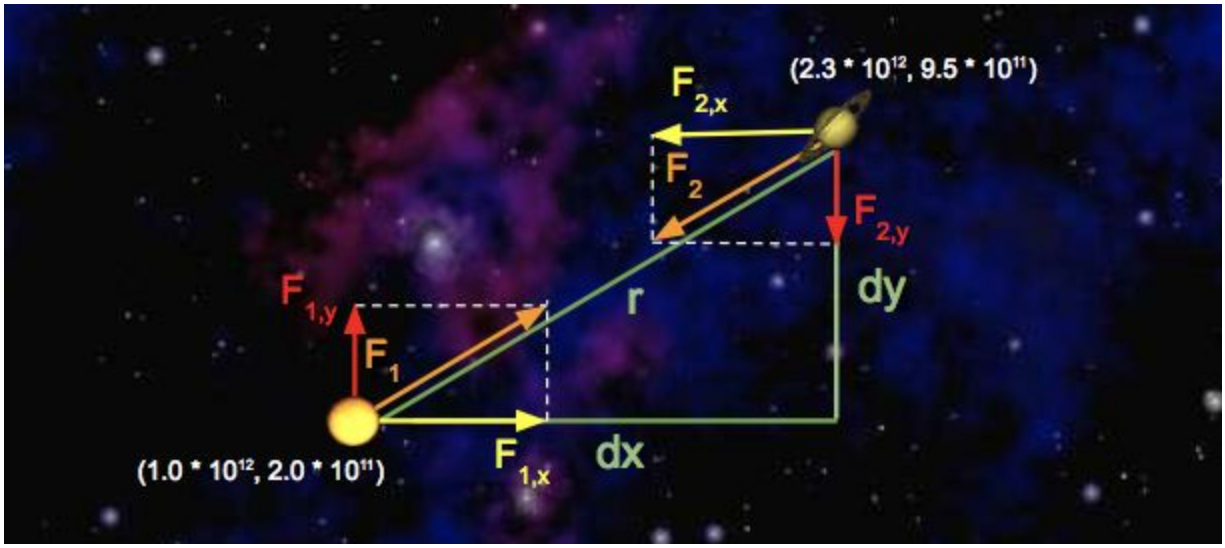
- **Acceleration**: Newton's *second law of motion* says that the accelerations in the x- and y-directions are given by the equations below (derived from F = ma):

  - $a_x = F_x/m$
  - $a_y = F_y/m$

Check your understanding!

Consider a small example consisting of two celestial objects: Saturn and the Sun. Suppose the Sun is at coordinates ($1.0 * 10^{12}$, $2.0 * 10^{11}$) and Saturn is at coordinates ($2.3 * 10^{12}$, $9.5 * 10^{11}$).

Assume that the Sun's mass is 2.0 * 10^{30} kg and Saturn's mass is 6.0 * 10^{26} kg. Here's a diagram of this simple solar system:



Let's run through some sample calculations. First let's compute F_1, the force that Saturn exerts on the Sun. We'll begin by calculating r, which we've already expressed above in terms of *dx* and *dy*. Since we're calculating the force exerted by Saturn, *dx* is Saturn's x-position minus Sun's x-position, which is 1.3 * 10^{12} meters. Similarly, *dy* is 7.5 * 10^{11} meters.

So, $r^2 = dx^2 + dy^2 = (1.3 \times 10^{12}\text{m})^2 + (7.5 \times 10^{11}m)^2$. Solving for r gives us 1.5 * 10^{12} meters. Now that we have r, computation of F is straightforward:

- $F = G \times (2.0 \times 10^{30}\text{kg}) \times (6.0 \times 1026\ \text{kg})\ /\ (1.5 \times 10^{12}\text{m})^2 = 3.6 \times 10^{22}\text{N}$

Note that the magnitudes of the forces that Saturn and the Sun exert on one another are equal; that is, |F| = $|F_1|$ = $|F_2|$. Now that we've computed the pairwise force on the Sun, let's compute the x and y-components of this force, denoted with $F_{1,x}$ and $F_{1,y}$, respectively. Recall that *dx* is 1.3 * 10^{12} meters and *dy* is 7.5 * 10^{11} meters. So,

- $F_{1,x} = F_1 * (1.3 * 10^{12}\text{ m}) / (1.5 * 10^{12}\text{ m}) = 3.1 * 10^{22}\text{ N}$
- $F_{1,y} = F_1 * (7.5 * 10^{11}\text{ m}) / (1.5 * 10^{12}\text{ m}) = 1.8 * 10^{22}\text{ N}$

Caution: The signs of *dx* and *dy* are important! Here, *dx* and *dy* were both positive, resulting in positive values for $F_{1,x}$ and $F_{1,y}$. This makes sense if you look at the diagram: Saturn will exert a force that pulls the Sun to the right (positive $F_{1,x}$) and up (positive $F_{1,y}$).

Next, let's compute the x and y-components of the force that the Sun exerts on Saturn. The values of *dx* and *dy* are negated here, because we're now measuring the displacement of the Sun relative to Saturn. Again, you can verify that the signs should be negative by looking at the diagram: the Sun will pull Saturn to the left (negative *dx*) and down (negative *dy*).

- $F_{2,x} = F_2 * (-1.3 * 10^{12}\text{ m}) / (1.5 * 10^{12}\text{ m}) = -3.1 * 10^{22}\text{ N}$

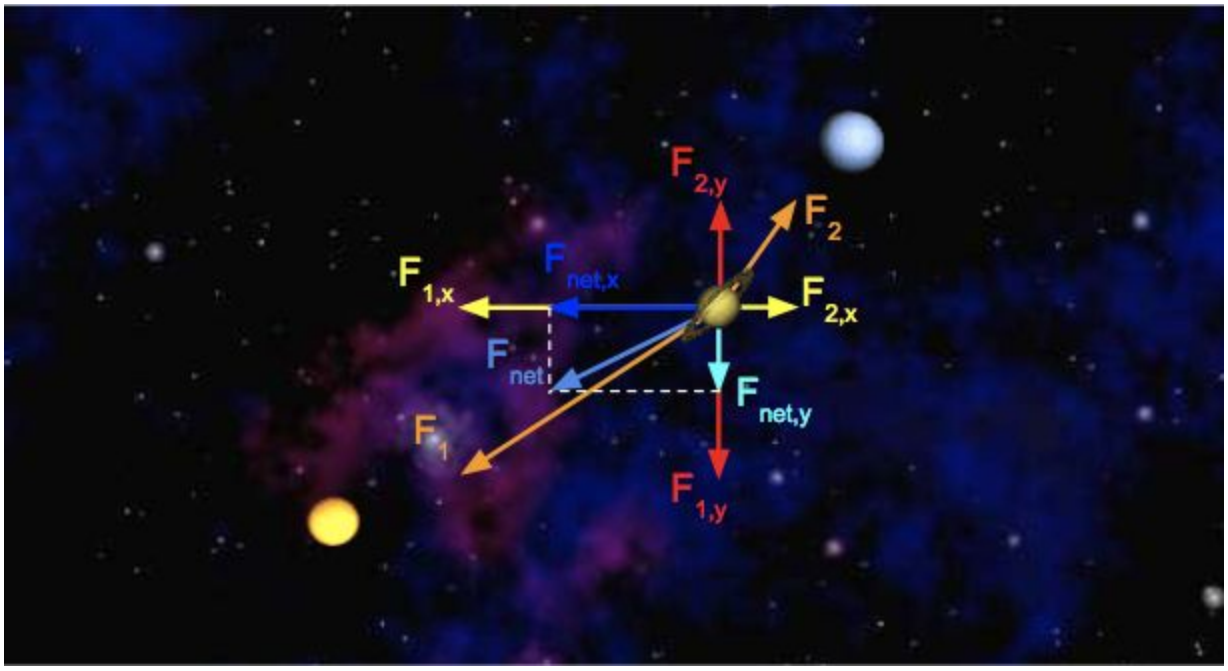
- $F_{2,y} = F_2 * (-7.5 * 10^{11} \text{ m}) / (1.5 * 10^{12} \text{ m}) = -1.8 * 10^{22} \text{ N}$

In short, when calculating the forces exerted **on** body A by another body B, we always use $(x_B - x_A)$ and $(y_B - y_A)$ to obtain *dx* and *dy*.

Below, you'll write the methods calcForceExertedByX and calcForceExertedByY in the Body class. When you're done with those methods, sun.calcForceExertedByX(saturn) and sun.calcForceExertedByY(saturn) should return $F_{1,x}$ and $F_{1,y}$, respectively; similarly, saturn.calcForceExertedByX(sun) and saturn.calcForceExertedByY(sun) should return $F_{2,x}$ and $F_{2,y}$, respectively.

Net Forces: Accumulating Forces From Celestial Bodies

Let's add Neptune to the mix and calculate the net force on Saturn. Here's a diagram illustrating the forces being exerted on Saturn in this new system:



We can calculate the x-component of the net force on Saturn by summing the x-components of all pairwise forces. Likewise, $F_{net,y}$ can be calculated by summing the y-components of all pairwise forces. Assume the forces exerted on Saturn by the Sun are the same as above, and that $F_{2,x} = 1.1 * 10^{22}$ N and $F_{2,y} = 9.0 * 10^{21}$ N.

- $F_{net,x} = F_{1,x} + F_{2,x} = -3.1 * 10^{22} \text{ N} + 1.1 * 10^{22} \text{ N} = -2.0 * 10^{22} \text{ N}$
- $F_{net,y} = F_{1,y} + F_{2,y} = -1.8 * 10^{22} \text{ N} + 9.0 * 10^{21} \text{ N} = -9.0 * 10^{21} \text{ N}$

Double check your understanding!

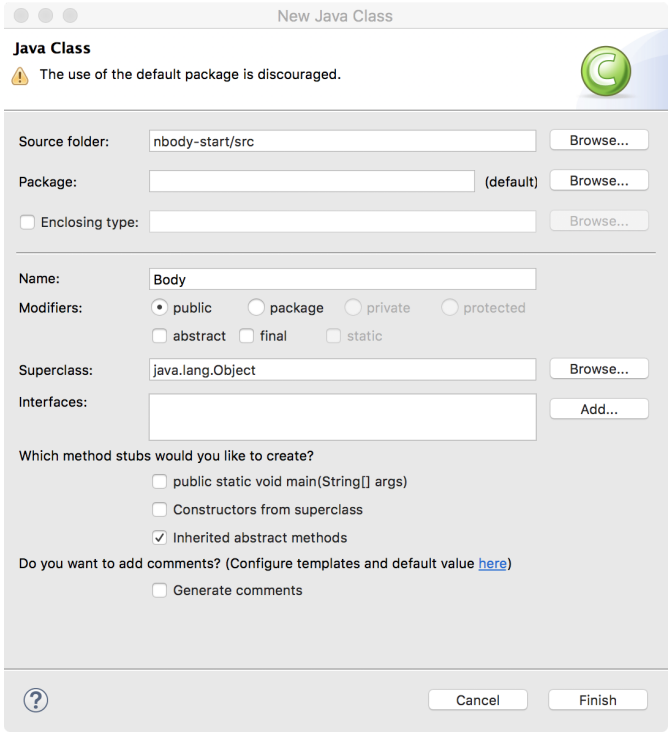
Suppose there are three bodies in space as follows:

- Samh: x = 1, y = 0, mass = 10
- AEgir: x = 3, y = 3, mass = 5
- Rocinante: x = 5, y = -3, mass = 50

Calculate $F_{net,x}$ and $F_{net,y}$ exerted on Samh. To check your answer, click here for the net x force and here for the net y force.

The Body Class

You'll start by creating a **`Body`** class. You'll need to create a new Java class named **`Body`**. You'll use *New>Class* in Eclipse to create the new class e.g., from the default package or the File menu. Make sure the class name is **`Body`** and that it's in the default (no name) package. This means that when you create the **`Body`** class you'll leave the field for *Package* blank. Eclipse will indicate that this is discouraged, as you can see in the screenshot below. It's ok!



Be sure you don't set a package name --- classes will be in the default/anonymous package for this project and most projects in 201. If Eclipse auto-generates a package name, click on the Browse button and pick default package.

Begin by creating a basic version of the **`Body`** class with the following 6 instance variables. *Make these variables have either no designation or public*. As shown below if you don't specify public or private you'll get package-level access which is fine here.

```
public class Body {

      double myXPos;            // current x position
      double myYPos;            // current y position
      double myXVel;            // current velocity in x direction
      double myYVel;            // current velocity in y direction
      double myMass;            // mass of body
      String myFileName;        // file name (in images folder)
```

Your instance variables must be named exactly as above.

Implementing Body Constructors

You must add two constructors: one that has parameters for all the instance variables and one that copies data from another **`Body`**. The signature of the first constructor should be:

```
public Body(double xp, double yp, double xv,
              double yv, double mass, String filename)
```

The second constructor should have a **`Body`** object as a parameter and initialize an identical **`Body`** object (i.e. a clone). The signature of the second constructor should be:

```
public Body(Body p)
```

Your **`Body`** class should NOT have a main method, because we'll never run the **`Body`** class directly. Also, the word "static" should not appear anywhere in your **`Body`** class.

All of the numeric values used in this project will be doubles. Unless otherwise specified, all methods will be public and all instance variables will be public or the default package access (with no designation).

Once you have filled in the constructors, you can test it out by running the **`main`** method in **`TestBodyConstructor`**. Running this class will save and compile your **`Body.java`** file and the **`TestBodyConstructor.java`** file we have provided. In general each method you write (and the constructor) can be tested using test classes we provide for this project. In later projects you'll be developing your own test classes and methods.

If you pass this test, you're ready to move on to the next step. **Do not proceed until you have passed the `TestBodyConstructor` test.**

Implementing Body Class Methods

In your program, you'll have instances of **`Body`** class do the job of calculating all the numbers you learned about in the previous example. You'll write helper methods, one by one, until your **`Body`** class is complete.

The helper methods here will be public, though in later examples we'll have occasion to make helper methods private. We're working to keep the first example simple in terms of cognitive-load, and there's lots of new material and concepts here.

`calcDistance`

Start by adding a method called **`calcDistance`** that calculates the distance between two Bodies using the following formula:

$$r^2 = dx^2 + dy^2$$

where *dx* is delta/difference between x-coordinates, similarly for *dy*.

This method should have one parameter: a body, and should return a double equal to the distance between the supplied body and the body that is doing the calculation, e.g., between **`otherPlanet`** the parameter, and this the object on which the **`calcDistance`** method is invoked.

In the example below, **`pp`** is one of the bodies and **`otherPlanet`** is the other body.

```
pp.calcDistance(otherPlanet);
```

Based on this context, you should be able to determine the *signature* of the method. Once you have completed this method, go ahead and run the testing program **`TestCalcDistance`** we've provided.

As a hint, you can see in the call to **`calcDistance`** that there is one parameter. What is its type? You're told what the return type is. That's enough to determine the method's signature.

Hint: Always try searching the web for before asking questions on Piazza. Knowing how to find what you want on Google is a valuable skill. However, know when to give up! If you start getting frustrated with your search attempts, turn to Piazza.

`calcForceExertedBy`

The next method that you will implement is **`calcForceExertedBy`**. The **`calcForceExertedBy`** method has one body as a parameter, and returns a double describing the force exerted on this body by the given body. You should calculate the force using the formula below:

$$F = G\frac{m_1 m_2}{r^2}$$

where m_1 and m_2 are the masses of the two bodies, and G is the gravitational constant ($6.67 * 10^{-11}$ N-m^2 / kg^2).

Note that you'll have two bodies: one is a parameter and the other is the object on which the **`calcForceExertedBy`** method is invoked -- so you have two mass values. For example

`pp.calcForceExertedBy(otherPlanet)` for the numbers in "Double Check Your Understanding" return $1.334 * 10^{-9}$. You should be calling the **`calcDistance`** method in this method.

Once you've finished writing **`calcForceExertedBy`** you can test it by running the provided test class in **`TestCalcForceExertedBy`**.

NOTE: Do not use **`Math.abs`** to fix sign issues with these methods. This will cause issues later when drawing bodies.

`calcForceExertedByX` and `calcForceExertedByY`

The next two methods that you should write are **`calcForceExertedByX`** and **`calcForceExertedByY`**. Unlike the **`calcForceExertedBy`** method, which returns the total force, these two methods describe the force exerted in the X and Y directions, respectively.

You can obtain the x- and y-components from the total force using these formulas:

$$F_x = F\frac{dx}{r}$$

$$F_y = F\frac{dy}{r}$$

Once you've finished, you can run the next unit testing class. For example **`pp.calcForceExertedByX(otherPlanet)`** in "Double Check Your Understanding" should return $1.0672 * 10^{-9}$.

You'll call **`calcForceExertedBy`** in each of these two methods -- and use the return value in calculating the value to return as explained in the Understanding Physics section.

You can test this method by the (surprisingly named) **`TestCalcForceExertedByXY`** unit-test/testing class.

`calcNetForceExertedByX` and `calcNetForceExertedByY`

Write methods **`calcNetForceExertedByX`** and **`calcNetForceExertedByY`** that each have one parameter: an array of bodies. These methods calculate the net X and net Y force, respectively, exerted by all bodies in that array upon the **`Body`** on which the method is invoked. The net X and net Y forces are obtained by summing up the individual X and Y components of each force, respectively. (Explanation)

For example, consider the code snippet below:

```
Body[] allPlanets = {pp, onePlanet, otherPlanet};
pp.calcNetForceExertedByX(allPlanets);
pp.calcNetForceExertedByY(allPlanets);
```

The two calls here would return the values given in "Double Check Your Understanding."

As you implement these methods, remember that bodies cannot exert gravitational forces on themselves! Can you think of why that is the case (hint: the universe will possibly collapse in on

itself, destroying everything including you)? To avoid this problem, ignore any body in the array that is equal to the current body. To compare two bodies, use the **`.equals`** method.

Note that you wouldn't use the name **`pp`** when calling **`.equals`**, because the variable is not visible inside the **`calcNetForceExertedByY`** method itself -- instead you'll be using the reserved word **`this`** to refer to the current object on which the **`calcNetForceExertedByY`** method is called. For example, in the call **`pp.calcNetForceExertedByY(allPlanets)`** the object is named **`pp`**, but in the body of the method the name **`this`** is used. In my code I have this statement in the body of a loop:

```
if (! p.equals(this)) {
    sum += calcNetForceExertedByY(p);
}
```

When you are done go ahead and run tests using the **`TestCalcNetForceExertedByXY`** class.

The void method `update`

Next, you'll implement the method update that determines the acceleration due to forces exerted on the body, and then make changes to the body's velocity and position based on the acceleration. This method will be called during the simulation to update the body's position and velocity at small time-steps. For example, pp.update(0.005, 10, 3) would adjust the velocity and position if an x-force of 10 Newtons and a y-force of 3 Newtons were applied for 0.005 seconds.

This example may make clear the method signature for **`update`** (you can rename the parameters).

```
public void update(double seconds, double xforce, double
yforce)
```

You *must compute* the movement of the **`Body`** using the following steps:

1. Calculate the acceleration using the provided x and y force values (parameters), using the following formulas:
$$a_x = F_x/m$$
$$a_y = F_y/m$$
2. Calculate the X and Y components of the new velocity by using the acceleration and current velocity. Recall that acceleration describes the change in velocity per unit time, so the new velocity is ($v_x + dt * a_x$, $v_y + dt * a_y$).
3. Calculate the new position by using the new velocity values computed in step 2 and the current position. The new position is ($p_x + dt * v_x$, $p_y + dt * v_y$).

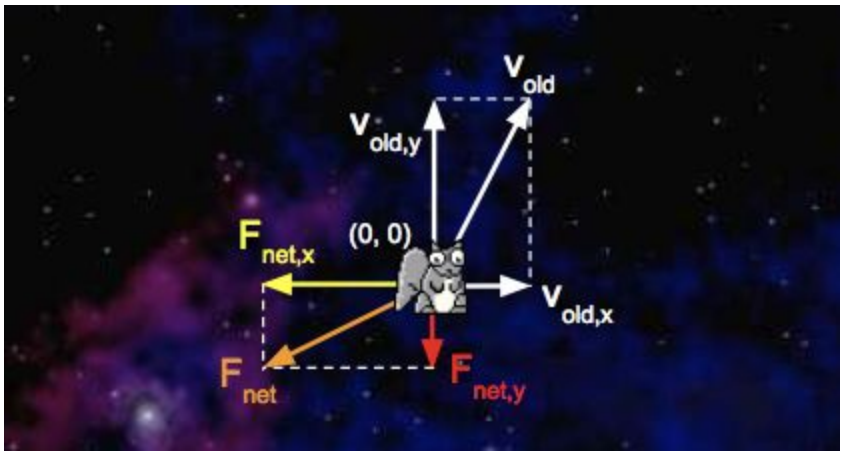
This means that update will determine new values of instance variables **`myXPos`**, **`myYPos`**, **`myXVel`**, and **`myYVel`** as explained in the steps above. The method has void return type because it is called to alter or mutate the instance variables and not to return a value.

Once you're done, test your method with by using the testing code in the class **`TestUpdate`**.

Once you've completed **`TestUpdate`**, you can move to the **`NBody`** class.

Example of how **`update`** works

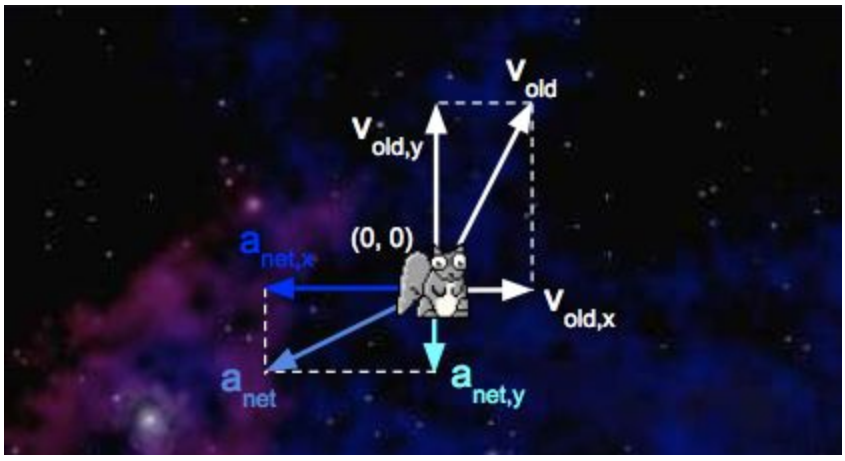
Let's try an example! Consider a (flying) squirrel initially at position (0, 0) with a v_x of 3 m/s and a v_y of 5 m/s. $F_{net,x}$ is -5 N and $F_{net,y}$ is -2 N. Here's a diagram of this system:



We'd like to update with a time step of 1 second. First, we'll calculate the squirrel's net acceleration:

- $a_{net,x} = F_{net,x} / m = -5 \text{ N} / 1 \text{ Kg} = -5 \text{ m/s}^2$
- $a_{net,y} = F_{net,y} / m = -2 \text{ N} / 1 \text{ Kg} = -2 \text{ m/s}^2$

With the addition of the acceleration vectors we just calculated, our system now looks like this:



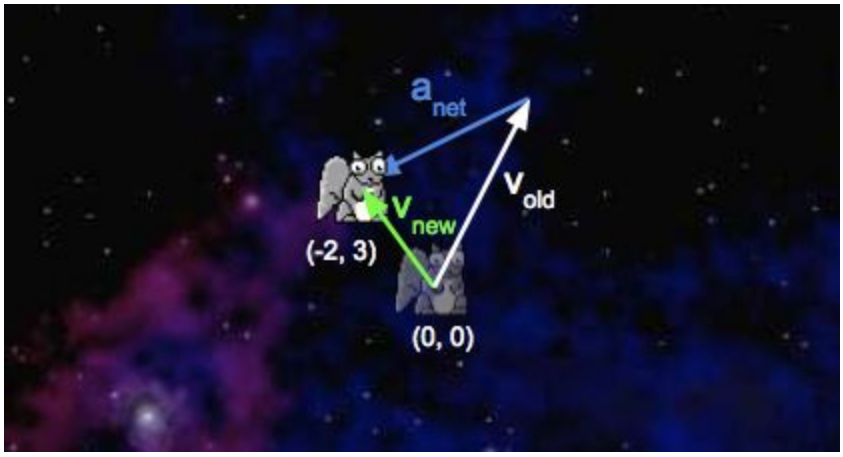
Second, we'll calculate the squirrel's new velocity:

- $v_{new,x} = v_{old,x} + dt * a_{net,x} = 3 \text{ m/s} + 1 \text{ s} * -5 \text{ m/s}^2 = -2 \text{ m/s}$
- $v_{new,y} = v_{old,y} + dt * a_{net,y} = 5 \text{ m/s} + 1 \text{ s} * -2 \text{ m/s}^2 = 3 \text{ m/s}$

Third, we'll calculate the new position of the squirrel:

- $p_{new,x} = p_{old,x} + dt * v_{new,x} = 0 \text{ m} + 1 \text{ s} * -2 \text{ m/s} = -2 \text{ m}$
- $p_{new,y} = p_{old,y} + dt * v_{new,y} = 0 \text{ m} + 1 \text{ s} * 3 \text{ m/s} = 3 \text{ m}$

Here's a diagram of the updated system:



For math/physics experts: You may be tempted to write a more accurate simulation where the force gradually increases over the specified time window. Don't! Your simulation must follow exactly the rules above.

The `NBody` Class

`NBody.java` will contain the class that will actually run your simulation. You'll start with a skeleton for **`NBody`** and implement several static methods in it to complete the simulation. This class will have NO constructor. The goal of this class is to simulate a universe specified in one of the data files. For example, if we look inside data/planets.txt as found here, we see the following:

```
5
2.50e+11
 1.4960e+11  0.0000e+00  0.0000e+00  2.9800e+04  5.9740e+24    earth.gif
 2.2790e+11  0.0000e+00  0.0000e+00  2.4100e+04  6.4190e+23     mars.gif
 5.7900e+10  0.0000e+00  0.0000e+00  4.7900e+04  3.3020e+23  mercury.gif
 0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  1.9890e+30      sun.gif
 1.0820e+11  0.0000e+00  0.0000e+00  3.5000e+04  4.8690e+24    venus.gif
```

The input format is a text file that contains the information for a particular universe (in SI units). The first value is an integer N which represents the number of bodies. The second value is a real number R which represents the radius of the universe, used to determine the scaling of the drawing window. Finally, there are N rows, and each row contains 6 values. The first two values are the x- and y-coordinates of the initial position; the next pair of values are the x- and y-components of the initial velocity; the fifth value is the mass; the last value is a **`String`** that is the name of an image file used to display the bodies. Image files can be found in the images directory. The file above contains data for our own solar system (up to Mars).

`readRadius`

Your first method is **`readRadius`**. Given a file name, it should return a double corresponding to the radius of the universe in that file, e.g. **`readRadius("./data/planets.txt")`** should return 2.50e+11.

The method signature is:

```
public static double readRadius(String fname)
```

You'll need to create a new **`Scanner`** from a new **`File`** object and you'll need to use try/catch as you've seen in examples for the code that initializes the **`Scanner`** object. Don't forget to close the **`Scanner`**! For example, my code looks something like this

```
public static double readRadius(String fname) {
    try {
        Scanner scan = new Scanner(new File(fname));
        // ...

        scan.close();
        return value;   // must return a double here
    } catch (FileNotFoundException e) {
        // print error message, call System.exit()
    }
```

You should be able to simply call **`scan.nextInt()`** to read the number of bodies (you can ignore the value in **`readRadius`**), then call **`scan.nextDouble()`** to complete the method **`readRadius`**.

We encourage you to do your best to use code examples from class as you figure out this part of the assignment on your own. In the long run, you'll need to gain the skills to independently figure out this sort of thing. However, if you start getting frustrated, don't hesitate to ask for help!

You can test this method using the supplied **`TestReadRadius`**.

`readBodies`

Your next method is **`readBodies`**. Given a file name, it should return an array of **`Body`** objects corresponding to the bodies in the file, e.g. **`readBodies("./data/planets.txt")`** should return an array of five bodies.

```
public static Body[] readBodies(String fname)
```

You will find the **`nextInt()`**, **`nextDouble()`**, and **`next()`** methods in the Scanner useful in reading int, double, and string values, respectively.

You can test this method using the supplied **`TestReadBodies`**. You should be sure to call this method in main to initialize the array of **`Body`** objects there.

Drawing the Initial Universe State (`main`)

Next, build the functionality to draw the universe in its starting position. You'll do this in four steps. Because all code for this part of the assignment is in **`main`**, this part of the assignment will NOT have automated tests to check each little piece.

Collecting All Needed Input

Modfiy the **`main`** method in the **`NBody`** class you start with so that the code will run the planetary/n-body simulation. Be sure you've verified that the code you get and the code you write (you'll need to add some code) will perform the following steps:

- Store the 0th and 1st command line arguments as doubles named **`totalTime`** and **`dt`**.
- Store the 2nd command line argument as a **`String`** named **`pfile`**.
- Read in the bodies and the universe radius from the file described by **`pfile`** using your methods from earlier in this assignment.

The first two steps are already done in the code you start with.

You should use variables local to the main method in case no values are supplied on the command-line. For default values you should use what's shown below (in the version of **`NBody`** you start with):

```
public static void main(String[] args) {

      double totalTime = 157788000.0;
      double dt = 25000.0;
      String pfile = "data/planets.txt";

      // more code here
    }
```

Drawing the Background

After your **`main`** method has read everything from the files, it's time to get drawing. First, set the scale so that it matches the radius of the universe. Then draw the image starfield.jpg as the background. To do these, you'll need to figure out how to use the **`StdDraw`** library. Here's code that might help with the size:

```
StdDraw.setScale(-radius, radius);
```

You should use **`StdDraw.picture`** to display a suitable background image. In my code I use

```
StdDraw.picture(0,0,"images/starfield.jpg");
```

In addition, you may want to check out the **`StdDraw`** section of this mini-tutorial, and if you're feeling bold, the full **`StdDraw`** documentation. This will probably take some trial and error. This may seem slightly frustrating, but it's good practice!

Drawing One `Body`

Next, we'll want a body to be able to draw itself at its appropriate position. To do this, take a brief detour back to the **`Body.java`** file to add a method to the **`Body`** class. Add method **`draw`**, that uses the **`StdDraw`** API mentioned above to draw the **`Body`**'s image (which is stored in instance variable **`myFileName`**) at the **`Body`**'s position. The new draw method should return nothing and have no parameters. Here's the call to the method in **`StdDraw`**:

```
StdDraw.picture(myXPos, myYPos, "images/"+myFileName);
```

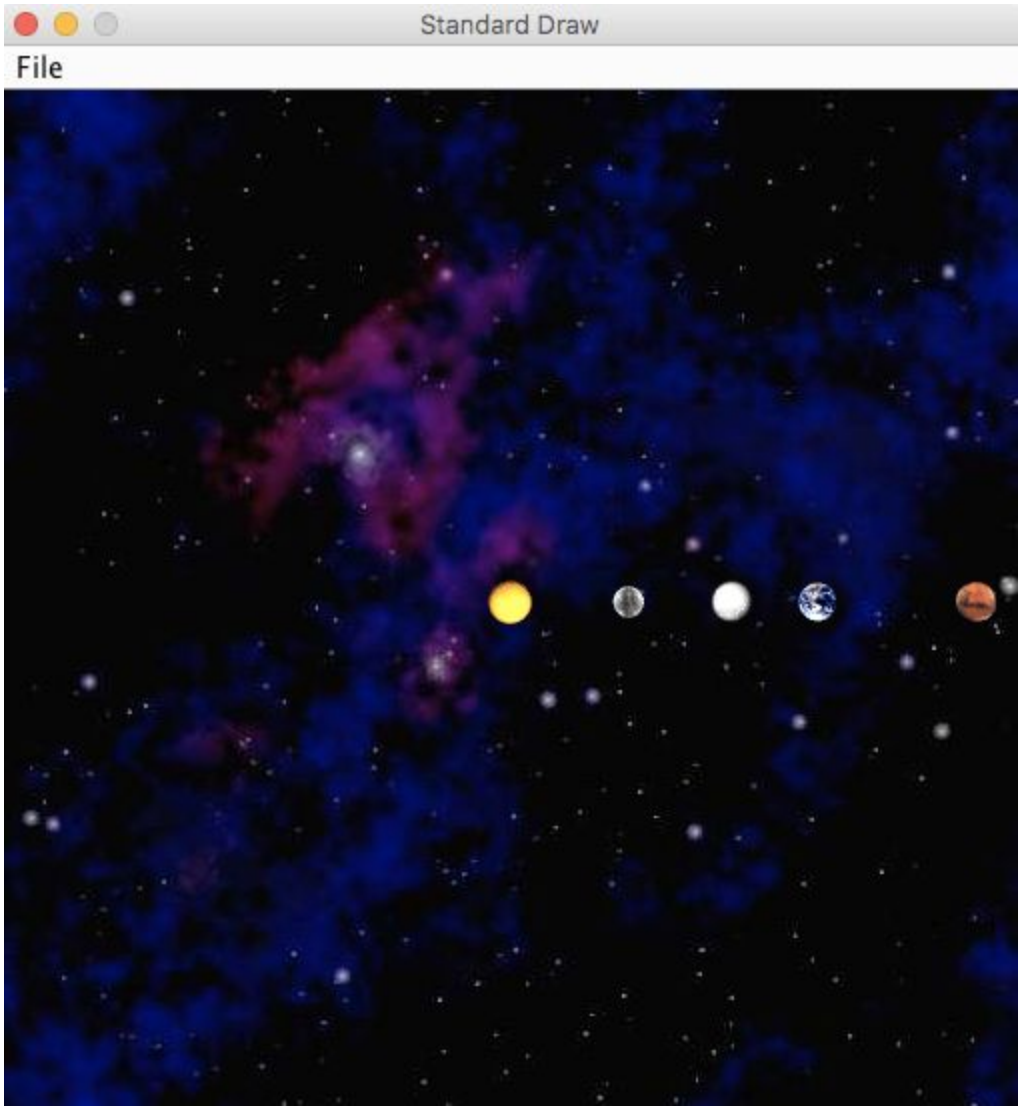
As shown, in the body of the draw method you'll call **`StdDraw.picture`** supplying the appropriate parameters based on instance variables of the **`Body`** object.

Drawing All of the Bodies

Return to the **`main`** method in **`NBody.java`** and use the **`Body.draw`** method you just wrote to draw each one of the bodies in the bodies array you created. Be sure to do this after drawing the starfield.jpg file so that the bodies don't get covered up by the background.

Test that your main method works by running **`Nbody`**. You can supply command line arguments to change the simulation's parameters, and we'll discuss how to do this in class and discussion, but you can use the default values described above rather than supplying command-line arguments.

You should see the sun and four bodies sitting motionless as shown below. You are almost done.



Creating an Animation

Everything you've done so far is leading up to this moment. With only a bit more code, we'll get something very cool.

To create our simulation, we will discretize time (please do not mention this to Stephen Hawking). The idea is that at every discrete interval, we will be doing our calculations and once we have done our calculations for that time step, we will then update the values of our Bodies and then redraw the universe.

Finish your main method by adding the following:

- Create a **`time`** (double) variable and set it to 0. Set up a **`for`** loop to iterate until this time variable is **`totalTime`**. You'll increment the **`time`** variable by **`dt`** on each loop iteration. This loop is the main body of the simulation.

- Each time through the loop, do the following:

- Create an **`xForces`** array and **`yForces`** array. Each should have the same capacity as the number of bodies in the simulation.
- (loop over bodies) Calculate the net x and y forces for each body, storing these in the **`xForces`** and **`yForces`** arrays respectively. You'll need to loop over bodies to do this, updating array entries in your loop. You'll call **`calcNetForceExertedByX`**, for example, to find values stored in the **`xForces`** array.
- (loop over bodies) Call update on each of the bodies. This will update each body's position, velocity, and acceleration. Again, you'll write a loop over bodies to do this. A separate loop after the previous one.
- Draw the background image.
- (loop over bodies) Draw all of the bodies.
- Pause the animation for 10 milliseconds (see the **`show`** method of **`StdDraw`**). You may need to tweak this on your computer. I use **`StdDraw.show(10)`** and get a good animation.
- Increase/increment your time variable by **`dt`**.

Important: For each time through the main loop, do not make any calls to update until all forces have been calculated and safely stored in **`xForces`** and **`yForces`**. For example, don't call **`bodies[0].update()`** until after the entire **`xForces`** and **`yForces`** arrays are done! The difference is subtle, but the autograder will be upset if you call **`bodies[0].update`** before you calculate **`xForces[1]`** and **`yForces[1]`**.

It's a good idea to use a for-loop over indexes in each step above labeled *(loop over bodies)*.

Compile and test your program by running **`NBody`**.

Make sure to also try out some of the other simulations, which can all be found in the data directory. Some of them are very cool.

Printing the Universe

When the simulation is over, i.e. when you've reached time **`totalTime`**, you should print out the final state of the universe in the same format as the input, e.g.:

```
5
2.50e11
 1.4925e+11 -1.0467e+10  2.0872e+03  2.9723e+04  5.9740e+24    earth.gif
-1.1055e+11 -1.9868e+11  2.1060e+04 -1.1827e+04  6.4190e+23     mars.gif
-1.1708e+10 -5.7384e+10  4.6276e+04 -9.9541e+03  3.3020e+23  mercury.gif
 2.1709e+05  3.0029e+07  4.5087e-02  5.1823e-02  1.9890e+30      sun.gif
 6.9283e+10  8.2658e+10 -2.6894e+04  2.2585e+04  4.8690e+24    venus.gif
```

The code for printing is given to you in the **`NBody.java`** you start with. This code isn't all that exciting (which is why we've provided a solution), but we'll need this method to work correctly to autograde your assignment. You should only print *after* your simulation completes.

REFLECT & Analysis

You should edit the *REFLECT.txt* file to include the necessary information. You will answer the basic questions about the assignment in the REFLECT.txt document that has been started for you. You should edit the file to replace the information in the square brackets with your information.

- Your name and NetID
- Hours Spent: Give the date you started the assignment, the date you completed the assignment, and an estimate of the number of hours you worked on it
- Consulted with: A list of the students, TAs, and professors with whom you consulted on the assignment. Since assignments are to be your own work, you should keep track of anyone with whom you have had a significant conversation about a program. You are welcome to talk with the course staff about the assignment, and to other students about broad ideas and concepts. If you did not consult with anyone, you *must* explicitly state that fact.
- Resources used: Note any books, papers, or online resources that you used in developing your solution. If you did not use any outside resources, you *must* explicitly state that fact.
- Impressions: You may include your impressions of the assignment to help the course staff improve it in the future.
- Answers to questions:
    1. What is the final position of the bodies (using the planets.txt file) after 1,000,000 and 2,000,000 seconds with a timestep of 25,000?
    2. Run the simulation with data/planets.txt and both a large value for totalTime (a billion or 10^9 works) and a large value for dt (a million or 10^6 works). You should observe behavior that isn't consistent with what celestial bodies should do. This behavior is due to using large values for dt in approximating the forces that act on the bodies. Write down what behavior you see in the simulation and a sentence or two in which you explain as best you can why large values don't result in accurate simulations.

If your answer has multiple lines, then you should use `%%%%` as a delimiter between answers. For example, a completed REFLECT.txt might look like the following.

```
Name: Sade Student
NetID: ss123
Hours Spent: 3.2
Consulted With: jforbes, gp70
Resources Used: Java API,
Stack Overflow post on constructors
https://stackoverflow.com/questions/579445/java-constructors
%%%%
Impressions: I thoroughly enjoyed checking out the various
data files. Who knew astrophysics could be so fun?!
%%%%
Question 1: What is the final position of the bodies after 1,000,000
seconds with a timestep of 25,000?
```

```
%%%%
Question 2: For what values of timeStep, does the simulation no longer behave
correctly?
%%%%
```

Submission & Grading

After you have run **`NBody`** on multiple data files and passed all of the tests, submit to the Project 0 : N-Body on Gradescope using the submit instructions. You may submit as many times as you'd like, but we will start restricting the autograder on future projects.

More details will be posted later.

This assignment is worth 20 points.

- 85% algorithmic/correctness: for the correctness of your implementation of NBody and Body (based largely on whether it passes our automated tests).
- 5% engineering: for the structure and style of your implementation. Does your solution decompose the problem appropriately? Is your code formatted appropriately?
- 10% analysis: for your REFLECT.txt and answers to the analysis questions.

Feel free to share your own custom universes on Piazza. Make sure to try out the other examples in the data folder!

Acknowledgements: We've been using the N-Body assignment at Duke since Spring 2008 as adapted from the assignment created by Robert Sedgewick and Kevin Wayne from Princeton University. This assignment has been adapted from a revision by Josh Hug, Matthew Chow, and Daniel Nguyen at Berkeley. Owen Astrachan updated Josh's version.

Frequently Asked Questions

- **I'm passing all the local tests, but failing even easy tests like `testReadRadius` in the autograder.**

Make sure you're actually using the string argument that **`testReadRadius`** takes as input. Your code should work for ANY valid data file, not just planets.txt.

- **The test demands 133.5, and I'm giving 133.49, but it still fails!**

Sorry, our sanity check tests are very particular. But you should ensure that your value for G is $6.67 * 10^{-11}$ N-m^2 / kg^2 exactly, and not anything else (don't make it more accurate).

- **When I run the simulation, my planets start rotating, but then quickly accelerate and disappear off of the bottom left of the screen.**

Look at the way you're calculating the force exerted on a particular body in one time step. Make sure that the force doesn't include forces that were exerted in past time steps.

Make sure you did not use **`Math.abs(...)`** when calculating **`calcForceExertedByX(...)`** and **`calcForceExertedByY(...)`**. Also ensure that you are using a double to keep track of summed forces (not int)!

In your simulation-for-loop of **`NBody`** be sure you have three separate steps:

1. First store in **`xForces`** and **`yForces`** the double values returned by **`calcNetForcExertedByX`** and **`calcNetForcExertedByY`** methods. One loop.
2. Call update on each body passing arrays calculated in previous step. One loop.
3. Draw all the bodies. One loop.

- **What is a constructor? How do I write one?**

A constructor is a block of code that runs when a class is instantiated with the new keyword. Constructors serve the purpose of initializing a new object's fields. Consider an example below:

```
public class Dog {
    String _name;
    String _breed;
    int _age;

    public Dog(String name, String breed, int age) {
        _name = name;
        _breed = breed;
        _age = age;
    }
}
```

The **`Dog`** class has three non-static fields. Each instance of the **`Dog`** class can have a name, a breed, and an age. Our simple constructor, which takes three arguments, initializes these fields for all new **`Dog`** objects.

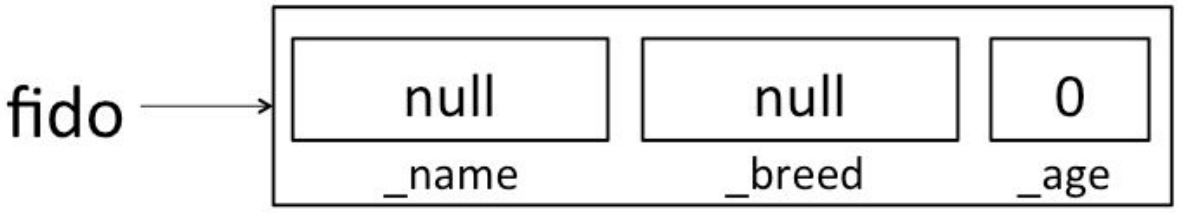
- **I'm having trouble with the second `Body` constructor, the one that takes in another Body as its only argument.**

Let's walk through an example of how a constructor works. Suppose you use the **`Dog`** constructor above to create a new **`Dog`**:

```
Dog fido = new Dog("Fido", "Poodle", 1);
```

When this line of code gets executed, the JVM first creates a new **`Dog`** object that's empty. In essence, the JVM is creating a "box" for the **`Dog`**, and that box is big enough to hold a box for

each of the **Dog**'s declared instance variables. This all happens before the constructor is executed. At this point, here's how you can think about what our new fluffy friend fido looks like (note that this is a simplification! We'll learn about a more correct view of this when we learn about objects and pointers later this semester):



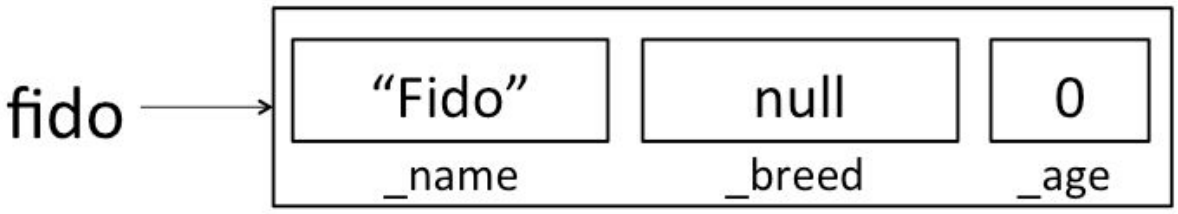
Java will put some default values in each instance variable. We'll learn more about where these defaults come from (and what null means) later this semester. For now, just remember that there's space for all of the instance variables, but those instance variables haven't been assigned meaningful values yet. If you ever want to see this in action, you can add some print statements to your constructor:

```
public Dog(String name, String breed, int age) {
    System.out.println("_name: " + _name + ", _breed: " + _breed + ", _age:
" + _age);
    _name = name;
    _breed = breed;
    _age = age;
}
```

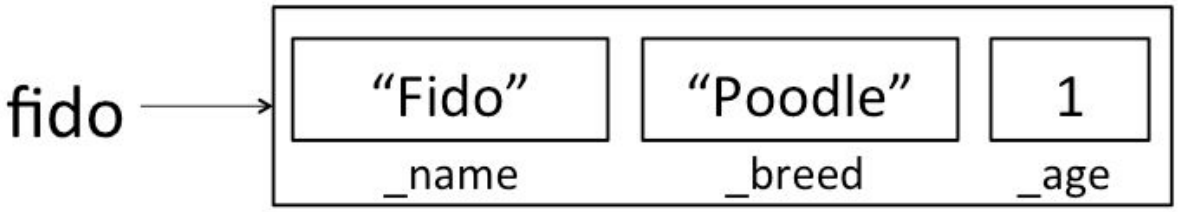
If this constructor had been used to create fido above, it would have printed:

```
_name: null, _breed: null, _age: 0
```

OK, back to making fido. Now that the JVM has made some "boxes" for fido, it calls the **Dog** constructor function that we wrote. At this point, the constructor executes just like any other function would. In the first line of the constructor, **_name** is assigned the value name, so that fido looks like:



When the constructor completes, fido looks like:



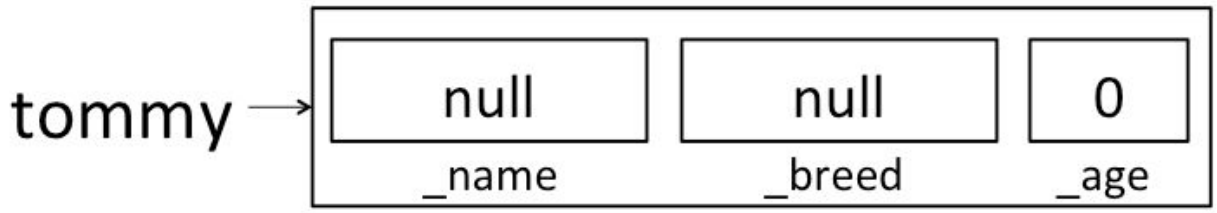
Now, suppose you want to create a new **`Dog`** constructor that handles cross-breeding. You want the new constructor to accept a name, an age, and two breeds, and create a new **`Dog`** that is a mixture of the two breeds. Your first guess for how to make this constructor might look something like this:

```
public Dog(String name, String breed1, String breed2, int age) {
    Dog newDog = new Dog(name, breed1 + breed2, age);
}
```

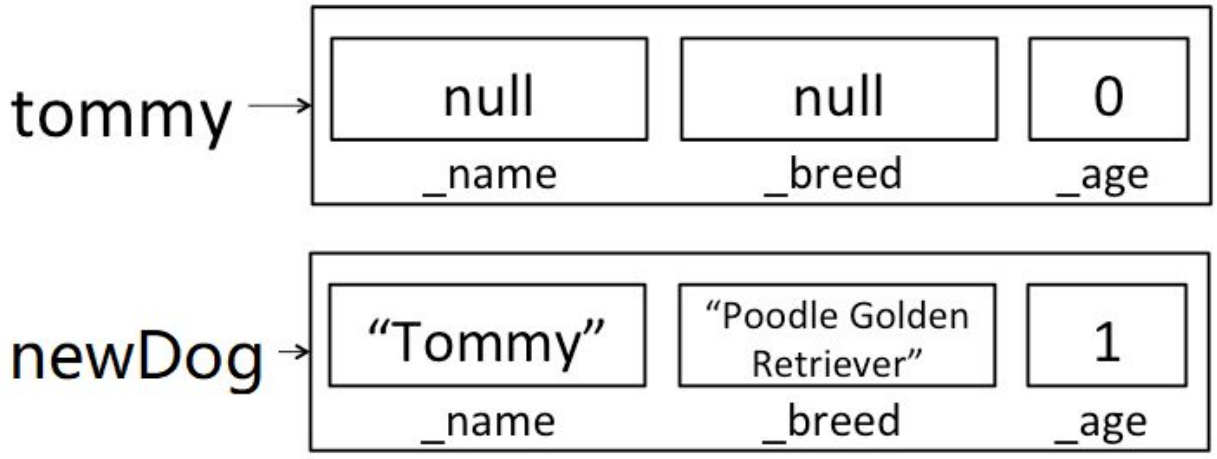
However, if you try to create a new **`Dog`** using this constructor:

```
Dog tommy = new Dog("Tommy", "Poodle", "Golden Retriever", 1);
```

This won't do what you want! As above, the first thing that happens is that the JVM creates empty "boxes" for each of tommy's instance variables:



But then when the 4-argument constructor got called, it created a second **`Dog`** and assigned it to the variable **`newDog`**. It didn't change any of **`tommy`**'s instance variables. Here's how the world looks after the line in our new constructor finishes:

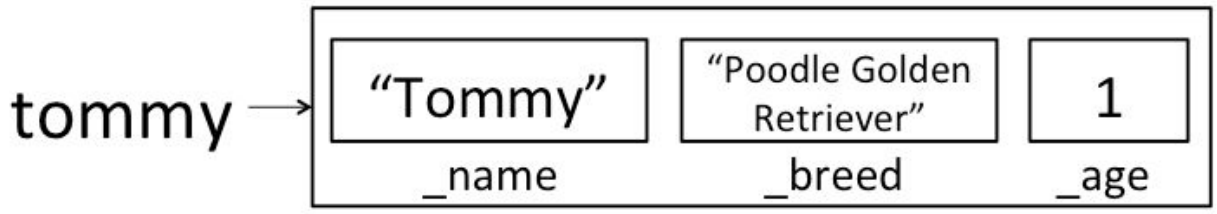


`newDog` isn't visible outside of the constructor method, so when the constructor completes, **`newDog`** will be destroyed by the garbage collector (more on this later!) and all we'll have is the still un-initialized **`tommy`** variable.

Here's a cross-breed constructor that works in the way we'd like:

```
public Dog(String name, String breed1, String breed2, int age) {
    Dog(name, breed1 + breed2, age);
}
```

Here, we're calling the old 3-argument constructor on this; rather than creating a new **`Dog`**, we're using the 3-argument constructor to fill in all of the instance variables on this dog. After calling this new constructor to create **`tommy`**, **`tommy`** will correctly be initialized to:



We could have also written a new constructor that assigned each instance variable directly, rather than calling the existing constructor:

```
public Dog(String name, String breed1, String breed2, int age) {
        _name = name;
        _breed = breed1 + breed2;
        _age = age;
    }
```

- **How do I cope with exceptions when making a Scanner?**

We've seen examples of this in code in class. Creating a new **`File`** object generates an exception that must be handled -- caught or rethrown. We recommend catching (and punting for now)

```
Scanner scan = null;
try {
    scan = new Scanner(new File(fname));
} catch (FileNotFoundException e) {
    // printing really isn't handling, but ok
    e.printStackTrace();
}
```

- **How do I read command-line arguments**

We'll go over how to pass arguments into your program from Eclipse. The basic idea is to use the Run Configurations > Arguments Tab > Program Arguments then Apply and Run.

To process them, if the value of array parameter args.length is greater than 0, you have String arguments passed in. Process them.

- **How do I use the `this` keyword?**

The keyword **`this`** refers to the current version of the object. For example, it's common to use **`this`** when parameters to a method have the same name as instance variables of the class.

```
public Point{
    int x;
    int y;

    public void setPosition(int x, int y){
        this.x = x;
        this.y = y;
    }
}
```

In the code above, the **`this`** keyword is used to differentiate between the x and y variables that belong to the **`Point`** class, and the x and y variables that are passed in in the method call. **`this`** can also be used on its own, to refer to the entire current instance of the object.

- **My constructor prints out all zeros!**

Make sure you are assigning variables correctly in your constructors. Where should you be storing values?

- **How do I read the datafile to make Bodies?**

What values do you need to create a **`Body`** object? Where are they inside the text file (you can open it in Eclipse or any text editor to check)? How do you get them out?

- **What is a static method/variable?**

Static methods and variables belong to a class, rather than to an instance of the class. This means that they must remain constant (static) throughout every incarnation of the class. To access them, you use the syntax [class name].method() instead of [object name].name(). An example is the **`Math`** class, which consists of static methods. This is why we can say "**`Math.pow(a,b)`**" instead of having to create a **`Math`** object.

- **My calcForceExertedBy method returns infinity!**

Remember not to count yourself (**`this`**) in the `calcForceExertedBy` method.

- **I created a variable in one part of my program, but when I try to use it Java says "[variable name] cannot be resolved to a variable"!**

If you create a variable at the top of a class (not inside any method), it will be accessible by every method inside the class. If you create a variable at the top of a method, it will be accessible by the entirety of that method, but not by other methods. If you create a variable inside an if statement, or for or while loop, it will only be accessible within the brackets of that statement/loop, and not in the rest of the method. Class-level variables can be used by other classes, unless you declare them with the word "private" in front. For this project, we want class-level variables (for example, **`myXPos`** in the **`Body`** class) to be visible to other classes, so you should **not** use the word "private" when declaring your variables.

- **How do I access other bodies' parameters?**

Because the **`Body`** variables are public variables, you can access them by "dereferencing" a body object. That is, by using the "." operator as follows: **`p1.myXPos`**;

- **What is a `Scanner`?**

A **`Scanner`** object reads ("scans") from sort of input. For example, the code

```
Scanner scan = new Scanner(System.in)
```

creates a **`Scanner`** that will read input from the user. In **`NBody`**, we want to create a **`Scanner`** that will read a file. We do this with the code

```
Scanner scan = new Scanner(new File(filename))
```

You can move the **`Scanner`** forwards through the file with the commands **`scan.nextInt()`**, **`scan.nextDouble()`**, **`scan.next()`**, etc (for an int, double, and string, respectively).

- **How do method headers work?**

Method headers consist of the following :

[visibility modifier] [return type] [name] ([parameter1 type] parameter1 name, [parameter 2 type] parameter 2 name, ...)

For example:

```
public void update (double seconds, double xForce, double yForce)
```

This method header tells us that the method is public (can be seen by other classes), void (returns nothing), called **`update`**, and has three parameters, all of type double.

- **My program runs, but my bodies keep flashing!**

Make sure that your call to **`StdDraw.show(10)`** is not inside the update loop or the draw loop.

- **What is a try/catch block?**

See the section on the **`FileNotFoundException`** below.

- **Do I need comments?**

Yes, you should always comment your code. Comments are there to increase understanding of your code. They help you remember what your code does if you come back to it after a couple days, and they help TAs understand and debug your code more easily. You shouldn't feel the need to comment every line of your code, but it's good practice to leave basic explanatory comments in your methods.

Here are the formats for adding comments in Java:

For single-line comments, use **`//`** at the start of the comment. Everything to its right will then become comments and be ignored by the compiler:

```
Scanner scan = new Scanner(new File(fname));  // Creates a new Scanner
```

And for comments that span over multiple lines, put **`/*`** and **`*/`** around them:

```
/* The following method calculates the acceleration based on
the forces passed in as arguments, and then updates the x- and y-
coordinates of the Body. */
public void update(double seconds, double xforce, double yforce)
```

Translations of common java errors:

- **`NullPointerException`**

You are trying to access something that doesn't exist. Make sure that you have instantiated your objects using the **`new`** keyword before you use them. Remember that if you create an array of objects, you will have to use **`new`** for the array creation as well as the object creation, as seen below:

```
String[] str = new String[3];

str[0] = new String("a");
str[1] = new String("b");
str[2] = new String("c");
```

- **`InputMismatchException`**

This error means that an object was expecting an input of one type and received something else. If you get this error in your **`readBodies`** method, make sure you are using **`Scanner.nextInt()`** and **`Scanner.nextDouble()`** as appropriate.

Additionally, it is common when using Scanners to read from them in a while loop where the condition is **`scan.hasNext()`**. You **do not** want to use this code for **`readBodies`**. Constructing your code like this will keep reading from the file the whole file has been looked at. If you open one of the **`NBody`** text files, you will see that there is text that we do not want to consider (citations, descriptions, etc). You should not need a while loop, because you should already know how many times to call the **`Scanner`** (hint: how many bodies are there?).

- **`ArrayIndexOutOfBoundsException`**

You tried to access an array via an index outside the bounds of the array. For example, the code below is **incorrect** and will throw this error. Remember that the first index in an array is 0, so the last index is array.length-1.

```
for (int i = 0; i <= array.length; i++){
      // your code here
}
```

- **`IllegalArgumentException`** (___.gif not found = images/ to file name)

You have passed in an invalid parameter for the method you are trying to use. If this error is accompanied by "[name].gif not found", the source of your error may be that the file name needs to be specified with "images/[file name]", so that Java knows to look in the images folder for the file.

- **`FileNotFoundException`** (try/catch)

See above (How do I cope with exceptions when making a Scanner?).

- Syntax error, insert ";" to complete Statement

You are missing a semicolon somewhere.

- Syntax error, insert "}" to complete ClassBody
- Syntax error on token "}", delete this token
- Syntax error on token "}", { expected after this token

You are missing a bracket somewhere. Ctrl+Shift+F (Command+Shift+F on a Mac) will auto-format your code for you, which may make it easier to find which line is missing a closing bracket.

- This method must return a result of type *x*

Your method expects you to be returning a variable of type *x* and you are not doing so. Remember that the method must return something no matter what, so you may get this error if your return statement is inside an if statement (only returning if the if statement is true).

- Type mismatch: cannot convert from double to int

You are trying to give an int variable a double value. Java will not let you do this, because it means losing precision. Make sure that your **`scan.nextInt()`** and **`scan.nextDouble()`** calls are being stored in variables of the correct type.

- Unreachable code

When a method reaches a return statement, it immediately ends. Therefore, if you have code after the return statement, it will never be executed. Java recognizes that this is “unreachable code” and will throw an error.