Project 1: NBody

Background

This assignment heavily borrows from Princeton and Berkeley Computer Science and the work of Robert Sedgewick, Kevin Wayne and Josh Hug.

Context. In 1687, Isaac Newton formulated the principles governing the motion of two particles under the influence of their mutual gravitational attraction in his famous *Principia*. However, Newton was unable to solve the problem for three particles. Indeed, in general, solutions to systems of three or more particles must be approximated via numerical simulations.
For a more complete understanding of the Physics you can reference this document.

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.

Starter Code

First fork the starter code from GitLab under your own namespace, then clone it using the **`git clone`** command (details below): https://coursework.cs.duke.edu/201spring19/p1-nbody

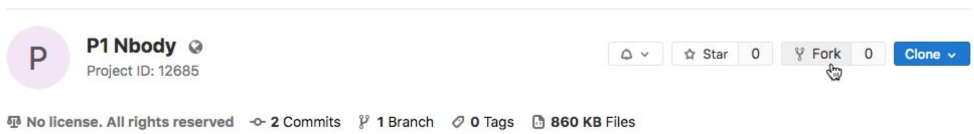
The starter code contains several image files in the **`images`** folder, data for several simulations in the **`data`** folder, the beginning of the **`NBody`** class in **`NBody.java`**, a stub version of **`CelestialBody.java`**, many testing Java files and library files for drawing. The analysis folder has questions you'll answer.

Note: You're given stub code (not complete) for all methods in the class **CelestialBody.java** -- as you complete these methods you'll get closer to completing the project.

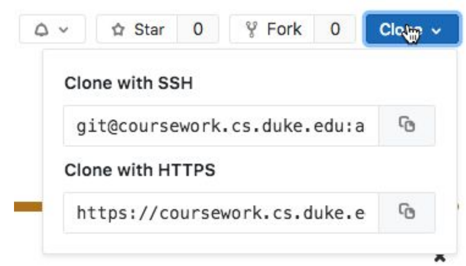
Integrating Git with Eclipse

Use these steps to clone the project and import to Eclipse:

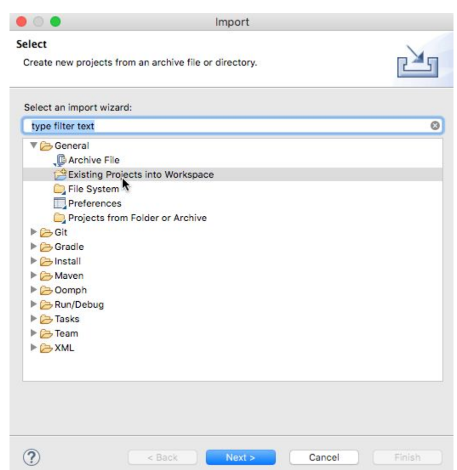
1. First fork the P1 NBody assignment using the link https://coursework.cs.duke.edu/201spring19/p1-nbody The screen shot below shows the fork icon.



2. Navigate in your Git shell using **cd** commands to your Eclipse workspace. Use **pwd** to verify you're there.
3. In your project's homepage on GitLab, you'll see the SSH URL for the project you'll use when you Clone the project from GitLab: see the image to the right.



   a. Copy the ***SSH URL*** using Command-C/Control-C
      - DO NOT use "Download ZIP"!
   b. In the Git shell, type **git clone <your-project-URL>**
      - Replace "**<your-project-URL>**" with the SSH URL you copied. Use Command-V/Control-V to paste.
   c. Typing '**ls**' should show a directory with the project name, which contains the files for the project.
4. Using the Eclipse File/Import menu, ***import an existing project into the workspace.*** You'll need to navigate to the folder/directory created when you used the **git clone** command. Use this for the project. DO NOT DO NOT import using the Git open. See the screen shot to the right. Using General>Existing Projects Into Workspace.



Pushing to Git

When you make a series of changes you want to 'save', you'll push those changes to your GitHub repository. You should do this after major changes, certainly every hour or so of coding. You'll need to use the standard Git sequence to commit and push to GitHub:

```
git add .
git commit -m 'a short description of your commit here'
git push
```

High-Level TODOs for Programming

- You'll create a class **`CelestialBody.java`** as described below. This class represents a Celestial Body such as a planet or a sun. You'll implement constructors, methods to get the state of a **`CelestialBody`** (getters), and methods that determine the interactions between **`CelestialBody`** objects due to gravitational forces. There are classes provided that help you test whether your constructors, getters, and interaction methods are correct.
- You'll create a class **`NBody.java`** that drives a simulation between planets, suns, and celestial bodies interacting. This class will read a file of data that specifies the initial positions and masses of the bodies and then simulates there interaction over a set time period. The simulation will also animate the interactions between the bodies.

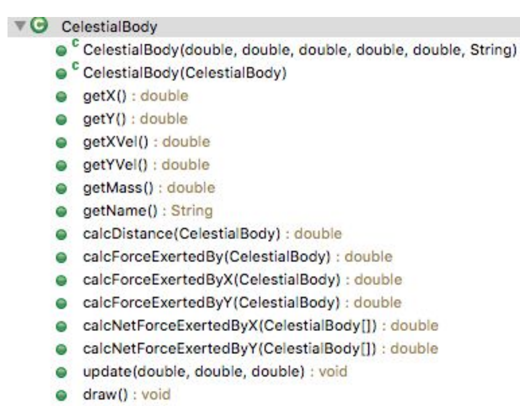
Details

You'll implement the **`CelestialBody`** class as described below. You must complete all stub constructors and methods. You'll do this in six steps, some of which require reading about the gravitational, physical interactions between two **`CelestialBody`** objects. Each step has a class to test whether your code works -- tests that are a strong indication your code works correctly. ***Be sure you pass the tests in each step before proceeding to the next step.***

When your **`CelestialBody.java`** class passes all tests you'll write code to simulate the interactions between N bodies. This code will be in the class **`NBody.java`**. You'll have some testing code for this class, but testing and debugging will require running simulations to see if the visualization and output match expected results.

Implement the `CelestialBody` class

The outline from Eclipse to the right shows all the instance variables, constructors, and methods of the **`CelestialBody`** class.



All instance variables should be **`private`**. All methods should be **`public`** (if you write helper methods they should be **`private`**).

Instance variables, Constructors, Getters

You'll have six instance variables: **`myXPos`**, **`myYPos`**, **`myXVel`**, **`myYVel`**, **`myMass`**, **`myFileName`**. The first five have type **`double`**, the last is a **`String`**.

You'll have two constructors: one in which values for each instance variables are given as parameters, and one that creates a **`CelestialBody`** by copying another. The signatures of these constructors are shown below.

```
/**
 * Create a Body from parameters
 * @param xp initial x position
 * @param yp initial y position
 * @param xv initial x velocity
 * @param yv initial y velocity
 * @param mass of object
 * @param filename of image for object animation
 */
public CelestialBody(double xp, double yp, double xv,
                     double yv, double mass, String filename){
    // TODO: complete constructor
}
```

```
/**
 * Copy constructor: copy instance variables from one
 * body to this body
 * @param b used to initialize this body
 */
public CelestialBody(CelestialBody b){
    // TODO: complete constructor
}
```

You'll also write six so-called ***getter*** methods specified in the diagram above. The body of each method is a single return statement, returning the value of the corresponding instance variable. These getter methods allow the values of private instance variables to be accessed outside the class. For example, the method **`getYVel()`** is shown to the right. These are *getter* methods because they do not allow client programs to *set* the values, only to *get* the values.

```
public double getXVel() {
    // TODO: complete method
    return 0.0;
}
```

When you've implemented the two constructors and the six getter methods you should be able to run the program in **`TestBodyConstructor.java`** to see if your code is correct. When it reports that everything works you can proceed to the next step in implementing the **`CelestialBody`** class.

The method `CelestialBody.calcDistance`

This method returns the distance between two **`CelestialBody`** objects. Use the standard distance formula to determine the distance between this body (using **`myXPos`** and **`myYPos`**) and the CelestialBody object specified by the parameter. The distance is the value of *r* in the formula below where

```
/**
 * Return the distance between this body and another
 * @param b the other body to which distance is calculated
 * @return distance between this body and b
 */
public double calcDistance(CelestialBody b) {
```

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

where dx is delta/difference between x-coordinates, similarly for dy. Use **`Math.sqrt`** to calculate the return value.

Test your implementation by running the program in **`TestCalcDistance.java`**.

The method `CelestialBody.calcForceExertedBy`

This method calculates and returns the force

```java
public double calcForceExertedBy(CelestialBody p) {
```

exerted on this body by the body specified as the parameter. You should calculate the force using the formula below. You can read about the physics of this formula in the NBody Physics document.

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

Here m_1 and m_2 are the masses of the two bodies, G is the gravitational constant ($6.67 * 10^{-11}$ $N\text{-}m^2 / kg^2$), and r is the distance between the two objects. Call **`calcDistance`** to determine the distance. You can specify *G* as 6.67*1e-11 using scientific notation in Java.

When you've implemented this method, test it by running **`TestCalcForceExertedBy.java`**.

The methods `CelestialBody.calcForceExertedByX` and `calcForceExertedByY`

These two methods describe the force exerted in the X and Y

```java
public double calcForceExertedByX(Body p) {
```

directions, respectively. The signature of **`calcForceExertedByX`** is shown above; **`calcForceExertedByY`** has a similar signature.

You can obtain the x- and y-components from the total force using these formulas below, where F is the value returned by **`calcForceExertedBy`**, r is the distance between two bodies, and F_x and F_y are the values returned by **`calcForceExertedByX`** and **`calcForceExertedByY`**, respectively. Note that *dx* that *dy* in the formula are DeltaX and DeltaY, the difference between x and y coordinates, respectively.

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

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

Note: Be careful with the signs! 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. You likely do not need to worry about this if you simply use the formulas shown here.

You can read about the physics for these formula in the NBody Physics document. You can test them using the program in **`TestCalcForceExertedByXY.java.`**

Note added 1/29 at 11:45 AM. ***Mathematically F/r * dx is the same as F*dx/r. However, because of roundoff error these may not be the same computationally. You should use F*dx/r in your method.***

The method `CelestialBody.calcNetForceExertedByX` and `calcNetForceExertedByY`

This method returns the total/net force exerted on this body by all the bodies in the array

```
public double calcNetForceExertedByY(CelestialBody[] bodies) {
```

parameter. The principle of superposition (see Physics) says that the net force acting on a body by many bodies is the sum of the pairwise forces acting on the **`CelestialBody`**. So you'll need to sum the forces returned by **`calcForceExertedByX`** (or Y) in calculating the value to return.

```
for(CelestialBody b : bodies) {
    if (! b.equals(this)) {
```

You must make sure NOT to include the force exerted by a body on itself! The universe might collapse if an object attracted itself. If you loop over each element in array bodies, you'll need code like this to avoid summing the force of an object on itself. In the body of the **`if`** statement you'd write code to accumulate the sum of all forces exerted on this **`CelestialBody`** by the **`CelestialBody`** b.

You can test your code by running the program in **`TestCalcNetForceExertedByXY.java.`**

The method `CelestialBody.update`

This method is a so-called mutator. It doesn't return a value, but updates the state/instance variables of the **`CelestialBody`** object on which it's called.

```
public void update(double deltaT,
                   double xforce, double yforce) {
```

This method will be called during the simulation to update the body's position and velocity with small time steps (the value of the first parameter, **`deltaT`**). The values of parameter **`xforce`** and **`yforce`** are the net forces exerted on this body by all other bodies in the simulation. When you're calling the **`update`** method from **`NBody.java`**, you will determine the values of the arguments passed as these two parameters when calling **`calcNetForceExertedByX`** (or Y). In the formulas below the parameter **`xforce`** is F_x and **`yforce`** is F_y.

This method updates the instance variables **myXPos**, **myYPos**, **myXVel**, and **myYVel** in four steps. In the formulas below the parameter **xforce** is F_x and **yforce** is F_y.

1. First, calculate the acceleration using Newton's second law of motion where m is the mass of the **CelestialBody**. This creates two variables for acceleration in the x and y directions.

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

2. You'll then calculate values for new **myXVel** and **myYVel**, we'll call these **nvx** and **nvy** where the n is for new, using the relationship between acceleration and velocity, e.g., **nvx = myXVel + deltaT*ax**.

3. You'll use **nvx** (and a corresponding **nvy**) to calculate new values for **myXPos** and **myYPos** using the relationship between position and velocity, e.g., **nx = myXPos + deltaT*nvx.**

4. After you've calculated **nx,ny,nvx**, and **nvy** you'll assign these to the instance variables **myXPos**, **myYPos**, **myXVel**, and **myYVel**, respectively.

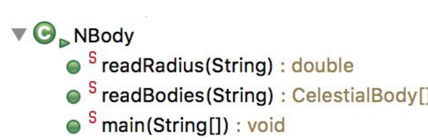
These steps will update the position and velocity of the body making the simulation possible. You can test this method using **TestUpdate.java**.

After developing, implementing, testing, and debugging these **CelestialBody** methods you're ready to move to the simulation code.

The method **CelestialBody.draw**

This **void** method is [described below](#) in the section for **NBody** that describes where to call the **CelestialBody.draw** method.

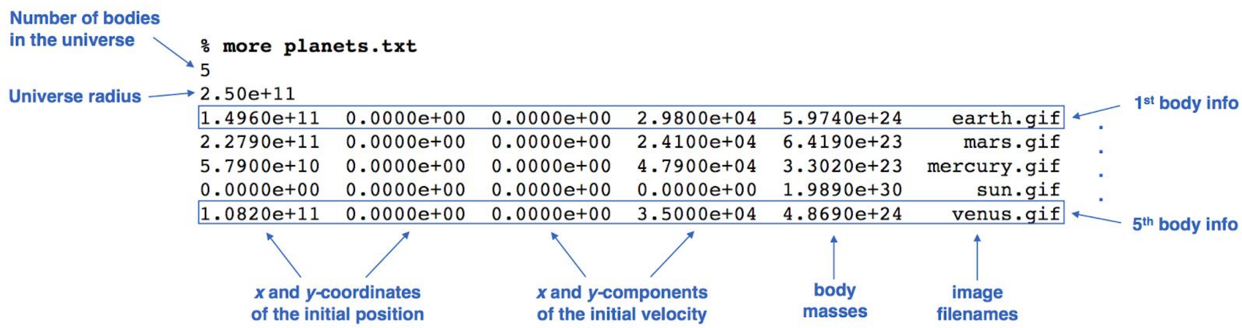
Implement the **NBody** class



This class consists only of **static** methods, including the main method that runs the simulation. Your task will be to implement the three static methods that have been started for you in the starter code you clone from Git. That code has **// TODO** comments indicating the code you need to add in the three static methods. These methods are described below.

File Format

The data for planets, suns, and celestial bodies in general is in the format shown below. All files in the folder data are in this format. This is the file planets.txt:



The first value is an integer ***n***, the number of bodies for which data is given in the file. The next value is a double, the radius of the universe for the simulation. This value is used to set the scale for the animation.

There are n lines, one line for each **`CelestialBody`**. Each line contains six values as shown above. The first five values are doubles: the first two are initial x and y coordinates; the next two are initial x and y velocities; the next is the mass of the **`CelestialBody`**. The last value on a line is a String specifying the file in the **images** folder used for the animation of the simulation.

The Method `NBody.readRadius`

Given a file name, this method 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. You'll need to read the **`int`** value that's the number of bodies, then read the **`double`** value for the radius. Use **`s.nextInt()`** and **`s.nextDouble()`** for the **`Scanner`** variable **`s`** to read an **`int`** and **`double`** value, respectively.

You can test your method using the provide **`TestReadRadius.java`** program.

The Method `NBody.readBodies`

This method returns an array of **`CelestialBody`** objects using the data read from the file. For example, **`readBodies("./data/planets.txt")`** should return an array of 5 **`CelestialBody`** objects.

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.java`** class. You should be sure to call this method in **`main`** to initialize the array of **`CelestialBody`** objects there.

The Method **`NBody.main`**

You'll see four TODO comments in the loop of the main program. Completing these will make your simulation run correctly and provide an animation of the simulation.

Completing the last TODO first will show a non-moving image for each body in the simulation. ***You'll write a for-each loop over each `CelestialBody` in the array `bodies`*** in the main method. In the loop you'll call the **`CelestialBody.draw`** method on each **`CelestialBody`** in the array. **You'll need to implement this method in the `CelestialBody` class** as shown above.

```
public void draw() {
    StdDraw.picture(myXPos, myYPos, "images/"+myFileName);
}
```

Most of the other TODOs in the loop will also be replaced by a loop, just as the drawing TODO used a loop over all the bodies.

- Create an **`xForces`** array and **`yForces`** array. Each should have the same size as the number of bodies in the simulation. You'll make new arrays on each iteration of the outer/simulation loop.
- (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 determine the values stored in the **`xForces`** array.
- (loop over bodies) Call **`update`** on each of the bodies. This will update each body's position and velocity. Again, you'll write a loop over bodies to do this. A separate loop after the previous one.

Printing the Universe

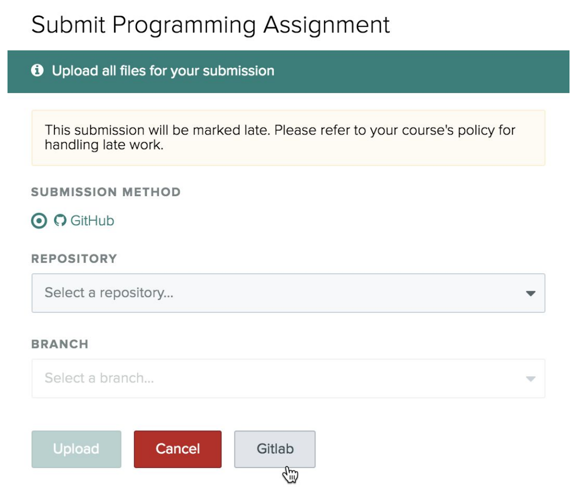
When the simulation is over your code prints out the final state of the universe in the same format as the input, e.g.:

```
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. ***You should NOT print anything other than the final printing shown here.***

Submitting

Push your code to Git. Do this often. You can use the autograder on Gradescope to test your code. UTAs will be looking at your source code to view documentation and your analysis.txt file, but you will be able to see the autograding part of your grade -- worth 16 points. Since you may uncover bugs from the autograder, you should wait until you've completed debugging and coding before completing the reflect form. You'll use the GitLab button to submit, shown in the screenshot to the right.



Complete the reflect after you've tested your code in Gradescope.

Reflect Form

[Complete the reflect form linked here] after submitting to Gradescope!

Analysis

You're given a folder named analysis, and in the folder there is a text file named analysis.txt. You should open that file and answer the questions in it by running your simulation program using the parameters provided in analysis.txt and copying/pasting the results as asked for. When you push via git, this folder and file will be pushed too. Undergrad TAs will look at your responses and grade them. Answer the questions in the analysis.txt file, they're reproduced here:

```
This is analysis.txt
Replace this line with your name
Replace this line with your netid

Copy/paste the output of your simulation when using planets.txt,
running the simulation for 1,000,000 (one  million) seconds, and
with a time-step/dt value of 25,000

(replace with copy/paste)
```

Copy/paste the output of your simulation when using planets.txt, running the simulation for 2,000,000 (two million) seconds, and with a time-step/dt value of 25,000

(replace with copy/paste)

Run the simulation for a billion seconds (10^9) and a time-step/dt of a million. You should see behavior inconsistent with what is expected for celestial bodies. This is due to large values of dt when approximating forces. Write down below what you see during this simulation.

(replace with observed behavior)

Grading and Javadoc Comments

Each method you write in the **`CelestialBody`** class should have a Javadoc comment. You can have the skeleton for such a comment generated automatically after you write the method header. With the cursor on the line above the header type **`/**`** and hit return. You'll see the outline generated and you can fill in the comment. See examples above given for the constructors and one of the getter methods. You'll see that some methods in the starter code already have comments. Use these as a guide. UTAs will look at your code and points may be deducted if you do not have Javadoc comments for every constructor and method you write.

The autograder will check all your **`CelestialBody`** methods and the **`NBody`** methods as well. You'll be able to see the results of running the tests when you submit to gradescope.

Points	Grading Criteria
16	Autograded constructor and methods for CelestialBody and code in NBody
6	Answers to analysis questions
2	Javadoc and code style

Grading will be done based on these cutoffs:

20-24: A
15-19: B

10-14: C
5-9: D