**2.1.3 Newton’s Lab – Helper Classes**

**Introduction**

In this assignment we will create more sophisticated object interactions than we did in Autumn. We will begin by looking at one of the most fundamental interactions between objects – gravitational force. In this scenario we will deal with very large bodies like planets and stars using Newton’s Law of Universal Gravitation.

If you are a little worried about dealing with Physics and mathematical formulas, don’t worry. This assignment will act as an introduction to help get your feet wet for gradual immersion into the concepts of Physics through Computer Science. This assignment does not go too deep into it – that is for later assignments!

Concepts covered are:

* Physics: Newton’s Law of Universal Gravitation
* Physics / Math: vectors
* Greenfoot helper classes
* Using double vars
* Abstract classes
* Overloading methods
* Overriding methods
* Default constructor
* Greenfoot Color class
* Keywords: **double, abstract, this, static, final,**

**Materials**

* Computer with Greenfoot

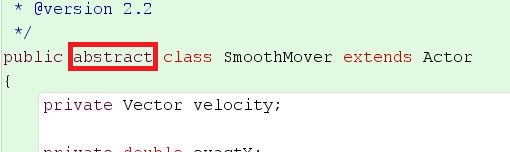
**Part I: Helper Classes & Overloading**

1. Download Newtons-Lab.zip and unzip into your project folder. Open the scenario. Take a look at the different classes and play with the scenario.
2. When you try to run the scenario you will find that you can place bodies into space, but they do not move or act in any interesting way…yet. Let’s explore:

|  |
| --- |
| * Invoke the various public methods of the Space object and see what they do. * When you have a star or planet in your world, right click it to see what public methods it has. * Invoke the sunPlanetMoon() method from the public methods of the Space object. Find and write down the mass of the sun, planet, and moon. You will need this later. * Look at the Space class source code and see how the public methods are implemented. |

1. This scenario contains two general purpose Greenfoot **helper classes** (programmers outside of Greenfoot normally call these **model libraries**). The helper classes are SmoothMover and Vector. What distinguishes a helper class is that it is never used to directly instantiate objects. Sometimes this means that the methods are static, something we’ve already encountered. In this case, however, the classes are **abstract**. These classes provide general functionality and can be used in multiple scenarios as you will see.

Examine the source code for SmoothMover. You will see the keyword **abstract** in the class signature.



Declaring a class **abstract** prevents instantiation of objects using the class. You will also notice that there is no constructor – abstract classes do not have them. Instead we create subclasses of abstract classes and the subclasses inherit the methods of the abstract class.

|  |
| --- |
| * Place an object of class Body (a subclass of SmoothMover) in the world and inspect it. Which methods did it inherit from SmoothMover? * Which method name appears twice? How do the versions differ? |

1. The SmoothMover class creates smoother movement for actors by storing the actor’s movement as doubles (64-bit decimal numbers) rather than integers. For displaying actors, the movement must still be rounded to integers for display, but the SmoothMover stores the exact decimal location of an object and still makes movement appear more streamlined. An actor instantiated from a SmoothMover subclass can, for example, have the x-coordinate 12.3 and can be moved along in increments of 0.6. Its successive location might be something like:

12.3, 12.9, 13.5, 14.1, 14.7… and so on

When it is being drawn it must be a whole pixel intervals so it would be at:

12, 13, 14, 14, 15…and so on

Altogether the effect is smoother-looking movement.

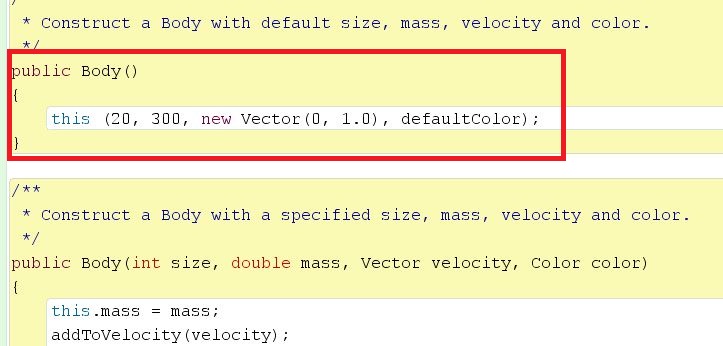
|  |
| --- |
| **Physics concept (velocity):**  The second part of the functionality of the SmoothMover is the use of a **velocity vector**. A velocity vector indicates the current speed and direction of movement. Negative velocity indicates movement in a direction that would decrease coordinate values. You can think of a vector as an arrow with a direction and a length proportional to its magnitude.  If you’ve never encountered this concept, the link below explains how velocity vectors work in sufficient detail for our purposes.  <http://www.physicsclassroom.com/class/1DKin/Lesson-1/Speed-and-Velocity>  An actor instantiated from a subclass of SmoothMover moves according to its velocity vector and can modify its movement by changing its velocity vector. |

1. The second helper class is Vector. This class implements the vector itself and is used by the SmoothMover. Note that Vector is not listed in the Actor group because it is not a subclass of Actor and will never occur in the world on its own. Objects of this class are only ever created by other objects and not directly.

|  |
| --- |
| **Physics concept(vectors):**  Vectors can represent more than just velocity – they can represent force, momentum, acceleration, and a variety of other physical characteristics. They can be represented mathematically in two ways: either as a pair of distances using x and y coordinates (dx,dy) or as a pair of values specifying the direction in degrees and magnitude as a length.  The first representation is called the Cartesian representation. The second is called the polar representation. You will see these two names inside the Vector class source code. For this program we will use both – whichever is easier for what we are doing – so our Vector class is written to handle both. It will do the necessary conversions internally and automatically.  Here’s a resource on using **polar vector calculations**:  <http://www.physicsclassroom.com/class/vectors/Lesson-1/Vectors-and-Direction>  Here’s a link showing **Cartesian vector** calculations:  <http://www.rasmus.is/uk/t/F/Su58k03.htm>  This requires use of the **Distance Formula** you learned in Math 8. If you don’t remember click this link:  <http://www.mathwarehouse.com/algebra/distance_formula/index.php> |

|  |
| --- |
| * Look at the documentation for Vector and SmoothMover and get to know the methods of each class. You do not need to look at the source code yet – it may be a bit confusing unless you understand the mathematical concepts well. To view documentation double-click the class diagram and then change the dropdown box in the upper right corner to Documentation. * Place a Body object into the world. Which of the methods inherited from SmoothMover can you call interactively (through the object’s menu)? Which can you not call at this stage? |

1. Open the source code for the Body class. Notice that he body class has two constructors, like some other classes you have seen before (bubbles for example). One of the constructors has no parameters – this we call the **default constructor** – and the other has four parameters.



If the same method name appears more than once in a class, we say that the method is **overloaded**. It is perfectly OK to have two of the same method within the same class as long as their parameter list is different (they have different signatures). When an overloaded method is called, the runtime system figures out which of the two is being used by examining the parameters we supply.

Let's also distinguish **overloading** with **overriding**, which we have done before though we did not use the term. We overload a method when we when we give two methods the same name within the same class. We **override** when we create a subclass and write a method that has the same name as one in the parent class. So....

* + Overloading & overriding are writing new methods with the same name as older methods
  + Overloading happens within one class, overriding happens when the new method is in a subclass
  + When a method is overloaded, both the old and new methods can be used by that class. When a method is overridden, only the latest method (in the subclass) can be used by that class

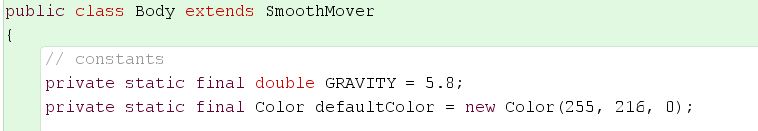
The default constructor makes it very easy to create a new Body object without having to specify all the details. The second, more complicated constructor, allows us to customize many aspects of the Body object. As you can see above, the default constructor uses the keyword **this** like it was a method call. This allows the default constructor to call the *other* constructor.

You’ll notice **this** in the second constructor as well. The Body class contains two variables called mass – one is declared as a field (instance variable) and the other used as a parameter. We need to use **this** to distinguish between the two. Here's the line in the second constructor:

**this**.mass = mass;

When we write mass without the keyword this, the nearest variable called mass is used (the parameter in this case). Adding the keyword allows us to use the field instead. So **this**.mass is an example of **overloading** a variable.

The last unusual portion of the Body class code is this:



The instance variables pictured above use the keywords **static** and **final**. We already know that **static** makes a variable or method ‘belong to the class’. So any object instantiated using the class will not inherit the variable. It also means that ALL the objects of the class *share* the variable instead of having their own copies of it. The **final** keyword makes a variable into a constant – meaning that it can never be reassigned or changed, only used in calculations. Also notice that, if the variable contains a value, we write the name in ALL CAPS!

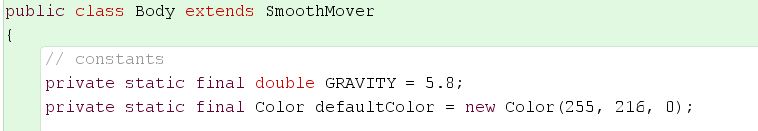
1. Enough playing – let’s make some objects move. You should have the source code for Body open:

|  |
| --- |
| * Add a call to the default move() method into the act() method. * Test it. * Examine the Body object: what is the default speed and direction? * Create multiple Body objects. How do they behave? * On the world screen, call all the public Space methods and run the scenario. How does it behave? How is the initial movement direction defined? * Change the default movement of Body to the left. Test. |

Currently each object has a velocity vector, but they do not interact and nothing changes the vector.

**Part II: Gravity’s Rainbow**

1. The second parameter of the Body class involves the use of the Color class in Greenfoot. At the top of the class the default color is defined:



The Color constructor takes 3 parameters that are numbers from 0 to 255. The first value represent red, the second represents green, and the third represents blue. When the three light colors are mixed we get the resulting color. For example: **new** Color(255,0,0) would produce the brightest possible red color.

|  |
| --- |
| * Look at the documentation for class Color – how many constructors does it have? * Change the default color to something else. Test it out. * Create three body objects of different colors on the world screen. |

1. It’s time to add gravitational interactions. Open the Body class source code and create a new method called applyForces().Here’s pseudocode for our task. Leave the higlighted part, applying gravity, empty for now and write the rest:

|  |
| --- |
| apply forces from other bodies:  get all other bodies in space:  for each of these bodies:  {  apply gravity from that body to this body;  }  Hint: Look at the code for Autumn – this is very similar to what we did to change the images of all the leaves. |

1. Applying gravity requires that we know Newton’s Law of Universal Gravitation. Here’s the law:

|  |
| --- |
| **Physics concept (Law of Universal Gravitation):**  G represents the universal gravitational constant and is roughly equal to: 6.67408(31)×10−11 m3⋅kg−1⋅s−2 |

The code is already set up to account for units and to convert, so the value we are using is one that works with our other mathematics. We have already defined the universal gravitational constant as a field called GRAVITY, which we looked at previously.

Let’s quickly recall how to do math in Java

All we need to do is write a new method of Body that will calculate force for us, return it as a double, and then call the new method inside our applyForces() method. Let’s call the new method applyMyGravity(). The Cartesian vector method is the easiest way to go here.

You will have to use Java Math methods to accomplish this. Open the Java documentation and look up class Math. We will need to use a method that will allow us to take a square root. Find the method.

|  |
| --- |
| apply my gravity(Body otherBody):  //find the distance from this body to the other body  dx = get the difference in x coordinates  dy = get the difference in y coordinates  distance = use distance formula with dy and dx    //Calculate the acceleration of gravity on this body  force = use univ grav formula, GRAVITY is G  acceleration = find my new accel give force  //Adjust velocity vector  //I need to use Vector class methods for this  dv = new Vector (dx,dy)  set the length of my vector to acceleration  add this length to my velocity vector |

1. Let’s try out what we’ve built:

|  |
| --- |
| * Try the three initialization method from the Space object one more time. What do you observe? * Experiment with changes in gravity by altering G (GRAVITY at the top of the Body class) * Experiment with changes to the mass and/or initial movement of the bodies defined in the Space class. * Create some new set ups of stars and planets and see how they interact. Try to come up with a system that is stable. |

You will see rather quickly that it is difficult to come up with a stable configuration for a solar system. This is because the simulation that you just created does not take into account several factors in the creation of a solar system – this is a very simplistic system. To make the system more accurate we would have to calculate all forces before each movement. Also the formula does not do a very good job of calculating forces when objects are very close to one another. Gravitation formulas have since been updated by Einstein and theorists who expanded upon General Relativity. Even accounting for these problems, it would not be easy to create a stable system.

So how did the solar system come to such stable equilibrium? The short answer – Ask Mr Carter. The long answer? The solar system formed when lumps of matter slowly collided with one another over a long period of time. Some fell into the sun and others aggregated into planets, moons, and other bodies. Over time the orbits that developed were generally stable. That being said, the only way to create a proper simulation would be to model the aggregation of small, random lumps of matter. For this, however, we would need a much more detailed and complicated knowledge of physics and a lot more time!

**Conclusion Questions**

1. What distinguishes a helper class from a regular class?
2. Which methods do objects of class Body inherit from SmoothMover?

1. What are the two indicators that a class is abstract? How are methods from abstract classes called?
2. How do you know a default constructor by looking at the code?
3. Which method(s) is overloaded in class Body? How do you know?
4. How many constructors are there in class Color?
5. Explain the algorithm for applying gravity to an object and determining a velocity vector at a given point.