

Homework # 1

due Monday, January 30, 10:00 PM

In this assignment, you will implement simple ADTs for particles in a gravity simulation program. We provide the driver and the main content class. You need to implement ADTs for “points,” “vectors” and “particles.”

1 Point

A **Point** is a simple **immutable** class with two fields of type `double`. By “immutable,” we mean that all fields are declared `final`—they are not modified once the object is constructed. We will use a convention that private fields should have names preceded by an underscore, so the two fields should be named `_x` and `_y`.

The class provides the following operations:

`Point(double, double)` The constructor.

`x()` Returns the first field. (Traditionally, this method would be named `getX()`.)

`y()` Returns the second field.

`toString` Returns a string of the form (x, y) , for example “(3.0, -1.5)”.

`asAWT` Return an AWT point with integer coordinates which round the double fields.

`distance(Point)` Returns the distance to the other point from here. This is done by computing the magnitude of the vector between them (see next class).

2 Vector

A **Vector** represents a direction (in two dimensional space). We use it to represent velocities (directed movement) of particles. A **Vector** is implemented as an immutable class with two double fields (`_dx` and `_dy`) and the following operations:

`Vector()` Create the empty vector (no movement).

`Vector(double, double)` Create the vector with the given amounts.

`Vector(Point, Point)` Create the vector from the first point to the second. If the first point is $(1, 2)$ and the second point is $(3, 0)$, then the resulting vector should be $\langle 2, -2 \rangle$.

`dx()` Returns movement in “x” direction.

`dy()` Returns movement in “y” direction.

`toString()` Return the same sort of string as for class `Point`, except with angle brackets, e.g. “ $\langle 1.0, -2.5 \rangle$ ”.

`move(Point)` Apply this vector to the point returning a new point. If this vector is $\langle 2, -2 \rangle$ and the argument point is $(1, 2)$, then the resulting point should be $(3, 0)$.

add(Vector) Return the new vector which combines the direction of this vector with the parameter vector. Adding $\langle 2, -2 \rangle$ and $\langle 1, 1 \rangle$ is $\langle 3, -1 \rangle$.

scale(double) Return the new vector scaled by the parameter; scaling the vector \mathbf{v} by x is written $x\mathbf{v}$. If you scale by 2, the new vector will go twice as far in both directions. If you scale by -1 , you get a vector in the equal but opposite direction. If $\langle 2, -2 \rangle$ is scaled by -0.5 , the result is $\langle -1, 1 \rangle$.

magnitude() Return the magnitude of the vector—how big a change it represents. Written $|\mathbf{v}|$, the magnitude of a vector can be computed using the formula $|\langle x, y \rangle| = \sqrt{x^2 + y^2}$. For example, the magnitude of $\langle -3, 4 \rangle$ is 5.

normalize() A convenience method that returns a new vector which has the same direction but magnitude 1. It is computed by scaling the vector with the inverse of the magnitude:

$$\frac{1}{|\mathbf{v}|}\mathbf{v}$$

For example, the normalization of $\langle -3, 4 \rangle$ is the *unit* vector $\langle -0.6, 0.8 \rangle$. Normalization is undefined for the zero vector (that is, you don't need to handle that case).

3 Particle

A `Particle` is a *mutable* class with four fields:

```
_position a Point;
_velocity a Vector;
_mass a double indicating how "big" the particle is;
_color a color for rendering purposes.
```

The mass and color of a particle do not change (and should be declared as `final`.)

It provides the following operations:

`Particle(Point, Vector, double, Color)` Initialize a particle.

`getPosition` The obvious getter.

`getVelocity` Another obvious getter.

`move` Advance the position by the velocity, using the `move` operation of `Vector`.

`applyForce` Using Newton's law $\mathbf{F} = m\mathbf{a}$ rewritten as $\mathbf{a} = \frac{1}{m}\mathbf{F}$ (using `scale`), update the velocity $\mathbf{v}' = \mathbf{v} + \mathbf{a}$ (using `add`).

`gravForceOn(Particle)` Return the gravitational force (as a `Vector`) that this particle exerts on the other particle. Do not modify either this or the other particle! The result is the scaled unit vector $\frac{(Gm_1m_2)}{d^2}\mathbf{u}_{21}$ where G is the gravitational constant (for this simulation, use $G = 1$), m_i are the masses of the two particles, d is the distance between the two particles and \mathbf{u}_{21} is the normalization of \mathbf{v}_{21} , the vector from the second particle's position to that of the first.

This method can assume that the argument `particle` is not null and that it is at a different location. (If the two particles coincided, then $d = 0$ would cause the computation to blow up.) Your code should use methods of the `Vector` and `Point` ADTs to do most of the work. We will penalize code that calls the `x()` or `dx()` methods directly instead of using high-level methods.

`draw(Graphics g)` Draw the particle as a disk (filled oval) with radius equal to the square root of the mass at its AWT position in the graphics context.

4 Driver Programs

We provide two programs to test your ADTs with: one a unit test driver which consists of static methods in a class `Test`, and the other a gravity simulation reached through a class `Main`. The gravity simulation uses a graphical animation class called `ThreeParticles` that should be placed in the `edu.uwm.cs351` package along with the ADTs.

The unit test will by default print an error message for each failed test and count the number of tests passed and failed. If you fail a test in an earlier section, it will completely skip tests in later sections. (If `Point` isn't working, there isn't any sense in testing `Particle`.)

When debugging your program, it makes more sense to have the test driver stop on the first error found so that you can use Eclipse to go directly to the failed test. This behavior can be achieved by adding `-throw` as a "Program Argument" on the "Arguments" tab of the "Run Configuration" for `Test`.

5 Files

The two drivers you need to copy for this week's assignment can be found in the directory `CLASSHOME/src/homework1`. Here, and in all assignments, we use `CLASSHOME` as a shorthand for `/afs/cs.uwm.edu/users/classes/cs351`. If you want to have this definition work for you, you can add the following line at the end of the file `“.cshrc”` in your home directory:

```
setenv CLASSHOME /afs/cs.uwm.edu/users/classes/cs351
```

You may also find it useful to create a "symbolic link" to this directory from your home directory. My link is called `cs351`.

Your task is to write the following files:

```
edu/uwm/cs351/Point.java
```

```
edu/uwm/cs351/Vector.java
```

```
edu/uwm/cs351/Particle.java
```

These files must exist in your `homework/1` directory before the deadline (10:00 PM, Monday). It is preferable to have them in a subdirectory called "src" (in other words, the `homework/1` directory can be your Eclipse project directory).

6 Using Eclipse to Set up Your Homework

We recommend that you install Eclipse (free download available from <http://www.eclipse.org>) and OpenAFS (free download available from <http://www.openafs.org>) on a network capable computer, for example, a Windows 7 laptop.

Then, once you have authenticated and have AFS tokens (using Network Identity Manager on Windows 7, or using `kinit` and `aklog` on MacOSX/Linux), you should create a Java project in Eclipse with the following steps:

1. Choose “File>New>Java Project”
2. In the dialog box, use a name such as `AFS-Homework-1`, and unselect “use default location” and instead use:
 - Windows: `\\afs\cs.uwm.edu\users\classes\cs351\401\pantherid\homework\1`
The folder must start with *two* backslashes. If you use only one, then you will create your project on your local computer, not AFS, and it will get a zero when graded.
 - MacOSX/Linux: `/afs/cs.uwm.edu/users/classes/cs351/401/pantherid/homework/1`
3. Then click “Finish”
4. Then open the project and right click on the “src” cluster, and then choose “New>Package.”
5. Create a package called `edu.uwm.cs351` where most of your code will go. The order is exactly backward to what you may expect. Don’t scramble the parts.
6. Right click on “src” again, and choose “Import...” and select “General>File System.”
7. Input from directory:
 - Windows: `\\afs\cs.uwm.edu\users\classes\cs351\src\homework1`
 - MacOSX/Linux: `/afs/cs.uwm.edu/users/classes/cs351/src/homework1`
8. Click on the directory in the left pane (don’t check the box) and then check the boxes for `Main.java` and `Test.java` and then click “Finish.”
9. After this import is done, right click on the newly created package `edu.uwm.cs351` and again select “Import...” and import from the same directory again, but this time grab `ThreeParticles.java`

You will now be ready to program your homework. Every time you save a file, it will be saved in AFS where the instructors and the graders can access it.

If you find OpenAFS being too unreliable, you may create a second Java project in the default location and copy and paste files from the local project to the AFS project when you want to get feedback from instructors or to be graded. We don’t recommend this option because many people forget to copy in the latest copy of their files into AFS and then don’t get credit for their work.

In later assignments, we often distribute partially implemented “skeleton files” with names such as `Sequence.java.SKEL`. This should be imported into the correct package as with a Java file. Afterward the successful import, use “Refactor>Rename” to rename the file without the `.SKEL` suffix. We use this convention so that (1) it is clear that this file is incomplete and (2) so you do not accidentally import a skeleton file on top of a Java file you have been working on.

We will not be repeating these instructions for later assignments. Please keep a reference to them.