```
   public static void main(String[] args) {
      SimulationControl.createApp(new PlanetApp());
   }
}
```

The `Planet` class in Listing 5.3 defines the physics and instantiates the numerical method. The latter is the Euler algorithm, which will be replaced in Problem 5.2. Note how the argument to the `initialize` method is used. The `System.arraycopy(array1,index1, array2,index2,length` method in the core Java API copies blocks of memory, such as arrays, and is optimized for particular operating systems. This method copies `length` elements of `array1` starting at `index1` into `array2` starting at `index2`. In most applications, `index1` and `index2` will be set equal to 0.

Listing 5.3 A class that models the rate equation for a planet acted on by an inverse square law force.

```
package org.opensourcephysics.sip.ch05;
import java.awt.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.numerics.*;

public class Planet implements Drawable, ODE {
   // GM in units of (AU)^3/(yr)^2
   final static double GM = 4*Math.PI*Math.PI;
   Circle circle = new Circle();
   Trail trail = new Trail();
   double[] state = new double[5];     // {x,vx,y,vy,t}
   Euler odeSolver = new Euler(this); // creates numerical method

   public void doStep() {
      odeSolver.step();                    // advances time
      trail.addPoint(state[0], state[2]); // x,y
   }

   void initialize(double[] initState) {
      System.arraycopy(initState, 0, state, 0, initState.length);
      // reinitializes the solver in case the solver accesses data
      // from previous steps
      odeSolver.initialize(odeSolver.getStepSize());
      trail.clear();
   }

   public void getRate(double[] state, double[] rate) {
      // state[]: x, vx, y, vy, t
      double r2 = state[0]*state[0]+state[2]*state[2];     // r squared
      double r3 = r2*Math.sqrt(r2);                        // r cubed
      rate[0] = state[1];           // x rate
      rate[1] = (-GM*state[0])/r3; // vx rate
      rate[2] = state[3];           // y rate
      rate[3] = (-GM*state[2])/r3; // vy rate
      rate[4] = 1;                  // time rate
   }

   public double[] getState() {
      return state;
   }
```



```
   public void draw(DrawingPanel panel, Graphics g) {
      circle.setXY(state[0], state[2]);
      circle.draw(panel, g);
      trail.draw(panel, g);
   }
}
```

The `Planet` class implements the `Drawable` interface and defines the `draw` method as described in Section 3.3. In this case we did not use graphics primitives such as `fillOval` to perform the drawing. Instead, the method calls the methods `circle.draw` and `trail.draw` to draw the planet and its trajectory, respectively.

Invoking a method in another object that has the desired functionality is known as *forwarding* or *delegating* the method. One advantage of forwarding is that we can change the implementation of the drawing within the `Planet` class at any time and still be assured that the `planet` object is drawable. We could, for example, replace the circle by an image of the Earth. Note that we have created a composite object by combining the properties of the simpler `circle` and `trace` objects. These techniques of encapsulation and composition are common in object oriented programming.

Problem 5.2 Verification of `Planet` and `PlanetApp` for circular orbits

(a) Verify `Planet` and `PlanetApp` by considering the special case of a circular orbit. For example, choose (in astronomical units) $x(t=0)=1$, $y(t=0)=0$, and $v_x(t=0)=0$. Use the relation (5.11) to find the value of $v_y(t=0)$ that yields a circular orbit. How small a value of Δt is needed so that a circular orbit is repeated over many periods? Your answer will depend on your choice of differential equation solver. Find the largest value of Δt that yields an orbit that repeats for many revolutions using the Euler, Euler–Cromer, Verlet, and RK4 algorithms. Is it possible to choose a smaller value of Δt, or are some algorithms, such as the Euler method, simply not stable for this dynamical system?

(b) Write a method to compute the total energy (see (5.5)) and compute it at regular intervals as the system evolves. (It is sufficient to calculate the energy per unit mass, E/m.) For a given value of Δt, which algorithm conserves the total energy best? Is it possible to choose a value of Δt that conserves the energy exactly? What is the significance of the negative sign for the total energy?

(c) Write a separate method to determine the numerical value of the period. (See Problem 3.9c for a discussion of a similar condition.) Choose different sets of values of $x(t=0)$ and $v_y(t=0)$, consistent with the condition for a circular orbit. For each orbit determine the radius and the period and verify Kepler's third law. ■

Problem 5.3 Verification of Kepler's second and third law

(a) Set $y(t=0)=0$ and $v_x(t=0)=0$ and find by trial and error several values of $x(t=0)$ and $v_y(t=0)$ that yield elliptical orbits of a convenient size. Choose a suitable algorithm and plot the speed of the planet as the orbit evolves. Where is the speed a maximum (minimum)?

(b) Use the same initial conditions as in part (a) and compute the total energy, angular momentum, semimajor and semiminor axes, eccentricity, and period for each orbit.