```
            ball.step();
         }
         System.out.println("Results");
         System.out.println("final time = "+ball.t);
         // numerical result
         System.out.println("y = "+ball.y+" v = "+ball.v);
         // analytic result
         System.out.println("y analytic = "+ball.analyticPosition());
      } // end of method definition
} // end of class definition
```

**Problem 2.11 Inheritance**

(a) Run the `FallingParticleApp` class. How can you tell that the constructor of the superclass was called?

(b) Rewrite the `SHO` class so that it is a subclass of `Particle`. Remove all unnecessary variables and implement the abstract methods.

(c) Write the target class `SHOParticleApp` to use the new `SHOParticle` class. Use the `analyticPosition` and `analyticVelocity` methods to compare the accuracy of the numerical and analytic answers in both the falling particle and harmonic oscillator models.

(d) Try to instantiate a `Particle` directly by calling the `Particle` constructor. Explain what happens when you compile this program. ■

If you examine the console output in Problem 2.11a, you should find that whenever an object from the subclass is instantiated, the constructor of the superclass is executed as well as the constructor of the subclass. You also will find that an abstract class cannot be instantiated directly; it must be extended first.

**Exercise 2.12 Extending classes**

(a) Extend the `FallingParticle` and `SHOParticle` classes and give them names such as `FallingParticleEC` and `SHOParticleEC`, respectively. These subclasses should redefine the `step` method so that it first calculates the new velocity and then calculates the new position using the new velocity, that is,

```
public void step() {
   v = v - g*dt;          // falling ball
   y = y + v*dt;
   t = t + dt;
}

public void step() {
   v = v - k*x*dt;        // harmonic oscillator
   x = x + v*dt;
   t = t + dt;
}
```

Methods can be redefined (overloaded) in the subclass by writing a new method in the subclass definition with the same name and parameter list as the superclass definition.



(b) Confirm that your new `step` method is executed instead of the one in the superclass.

(c) The algorithm that is implemented in the redefined `step` method is known as the *Euler–Cromer* algorithm. Compare the accuracy of this algorithm to the original Euler algorithm for both the falling particle and the harmonic oscillator. We will explore the Euler–Cromer algorithm in more detail in Problem 3.1. ■

The falling particle and harmonic oscillator programs are simple, but they demonstrate important object-oriented concepts. However, we typically will not build our models using inheritance because our focus is on the physics and not on producing a software library, and also because readers will not use our programs in the same order. We will find that our main use of inheritance will be to extend abstract classes in the Open Source Physics library to implement calculations and simulations by customizing a small number of methods.

So far our target classes have only included one method, `main`. We could have used more than one method, but for the short demonstration and test programs we have written so far, such a practice is unnecessary. When you send a short email to a friend, you are not likely to break up your message into paragraphs. But when you write a paper longer than about a half a page, it makes sense to use more than one paragraph. The same sensitivity to the need for structure should be used in programming. Most of the programs in the following chapters will consist of two classes, each of which will have several instance variables and methods.

## 2.5 ■ THE OPEN SOURCE PHYSICS LIBRARY

For each exercise in this chapter, you have had to change the program, compile it, and then run it. It would be much more convenient to input initial conditions and values for the parameters without having to recompile. However, a discussion of how to make input fields and buttons using Java would distract us from our goal of learning how to simulate physical systems. Moreover, the code we would use for input (and output) would be almost the same in every program. For this reason input and output should be in separate classes so that we can easily use them in all our programs. Our emphasis will be to describe how to use the Open Source Physics library as a tool for writing graphical interfaces, plotting graphs, and doing visualizations. If you are interested, you can read the source code of the many Open Source Physics classes and can modify or subclass them to meet your special needs.

We first introduce the Open Source Physics library in several simple contexts. Download the Open Source Physics library from <www.opensourcephysics.org> and include the library in your development environment. The following program illustrates how to make a simple plot.

**Listing 2.8** An example of a simple plot.

```
package org.opensourcephysics.sip.ch02;
import org.opensourcephysics.frames.PlotFrame;

public class PlotFrameApp {
   public static void main(String[] args) {
      PlotFrame frame = new PlotFrame("x", "sin(x)/x", "Plot example");
      for(int i = -100;i<=100;i++) {
         double x = i*0.2;
```