```
      SimulationControl.createApp(new SimulationApp());
   }
}
```

Exercise 2.18 `AbstractSimulation` class
Run `SimulationApp` and see how it works by clicking the buttons. Explain the role of the various buttons. How many times per second is the `doStep` method invoked when the simulation is running? ■

The buttons in the `SimulationControl` that were used in `SimulationApp` in Listing 2.12 invoke methods in the `AbstractSimulation` class. These methods start and stop threads and perform other housekeeping chores. When the user clicks the Initialize button, the simulation's `Initialize` method is executed. When the Reset button is clicked, the `reset` method is executed. If you don't write your own versions of these two methods, their default versions will be used. After the Initialize button is clicked, it becomes the Start button. After the Start button is clicked, it is replaced by a Stop button, and the `doStep` method is invoked continually until the Stop button is clicked. The default is that the frames are redrawn every time `doStep` is executed. Clicking the Step button will cause the `doStep` method to be executed once. The New button changes the Start button to an Initialize button, which forces the user to initialize a new simulation before restarting. Later we will learn how to add other buttons that give the user even more control over the simulation.

A typical simulation needs to (1) specify the initial state of the system in the `initialize` method, (2) tell the computer what to execute while the thread is running in the `doStep` method, and (3) specify what state the system should return to in the `reset` method.

We could modify the falling particle model to use `AbstractSimulation`, but such a modification would not be very interesting because there is only one particle, and all motion takes place in one dimension. Instead, we will define a new class that models a ball moving in two dimensions, and we will allow the ball to bounce off the ground and off of the walls.

Listing 2.13 `BouncingBall` class.

```
package org.opensourcephysics.sip.ch02;
import org.opensourcephysics.display.Circle;

// Circle is a class that can draw itself
public class BouncingBall extends Circle {
   final static double g = 9.8;
   final static double WALL = 10;
   // initial position and velocity
   private double x, y, vx, vy;

   public BouncingBall(double x, double vx, double y, double vy) {
      this.x = x;   // sets instance value equal to passed value
      this.vx = vx; // sets instance value equal to passed value
      this.y = y;
      this.vy = vy;
      // sets the position using setXY in Circle superclass
      setXY(x, y);
   }

   public void step(double dt) {
      x = x+vx*dt; // Euler algorithm for numerical solution
```



```
      y = y+vy*dt;
      vy = vy-g*dt;
      if(x>WALL) {
         vx = -Math.abs(vx); // bounce off right wall
      } else if(x<-WALL) {
         vx = Math.abs(vx);  // bounce off left wall
      }
      if(y<0) {
         vy = Math.abs(vy); // bounce off floor
      }
      setXY(x, y);
   }
}
```

To model the bounce of the ball off a wall, we have added statements such as

```
if (y < 0) vy = Math.abs(vy);
```

This statement insures that the ball will move up if $y < 0$, and is a crude implementation of an elastic collision. (The `Math.abs` method returns the absolute value of its argument.)

Note our first use of the `if` statement. The general form of an `if` statement is as follows:

```
if (boolean_expression) {
   // code executed if boolean expression is true
} else {
   // code executed if boolean expression is false
}
```

We can test multiple conditions by chaining `if` statements:

```
if (boolean_expression) {
   // code goes here
} else if (boolean_expression) {
   // code goes here
} else {
   // code goes here
}
```

If the first boolean expression is true, then only the statements within the first brace will be executed. If the first boolean expression is false, then the second boolean expression in the `else if` expression will be tested, and so forth. If there is an `else` expression, then the statements after it will be executed if all the other boolean expressions are false. If there is only one statement to execute, the braces are optional.

The `BouncingBall` class is similar to the `FallingBall` class except that it extends `Circle`. We inherit from the `Circle` class because this class includes a simple method that allows the object to draw itself in an Open Source Physics frame called `DisplayFrame`, which we will use in `BouncingBallApp`. In the latter we instantiate `BouncingBall` and `DisplayFrame` objects so that the circle will be drawn at its x-y location when the frame is displayed or while a simulation is running.

To make the animation more interesting, we will animate the motion of many noninteracting balls with random initial velocities. `BouncingBallApp` creates an arbitrary number of noninteracting bouncing balls by creating an array of `BouncingBall` objects.