Listing 2.14 BouncingBallApp class.

```
package org.opensourcephysics.sip.ch02;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class BouncingBallApp extends AbstractSimulation {
   // declares and instantiates a window to draw balls
   DisplayFrame frame = new DisplayFrame("x", "y", "Bouncing Balls");
   BouncingBall[] ball; // declares an array of BouncingBall objects
   double time, dt;

   public void initialize() {
      // sets boundaries of window in world coordinates
      frame.setPreferredMinMax(-10.0, 10.0, 0, 10);
      time = 0;
      frame.clearDrawables(); // removes old particles
      int n = control.getInt("number of balls");
      int v = control.getInt("speed");
      // instantiates array of n BouncingBall objects
      ball = new BouncingBall[n];
      for(int i = 0;i<n;i++) {
         double theta = Math.PI*Math.random(); // random angle
         // instantiates the ith BouncingBall object
         ball[i] = new BouncingBall(0, v*Math.cos(theta), 0,
                        v*Math.sin(theta));
         // adds ball to frame so that it will be displayed
         frame.addDrawable(ball[i]);
      }
      // decimalFormat instantiated in superclass and used to format
      // numbers conveniently
      // message appears in lower right hand corner
      frame.setMessage("t = "+decimalFormat.format(time));
   }

   // invoked every 1/10 second by timer in AbstractSimulation
   // superclass
   public void doStep() {
      for(int i = 0;i<ball.length;i++) {
         ball[i].step(dt);
      }
      time += dt;
      frame.setMessage("t="+decimalFormat.format(time));
   }

   // invoked when start or step button is pressed
   public void startRunning() {
      dt = control.getDouble("dt"); // gets time step
   }

   public void reset() { // invoked when reset button is pressed
      // allows dt to be changed after initializaton
      control.setAdjustableValue("dt", 0.1);
      control.setValue("number of balls", 40);
      control.setValue("speed", 10);
   }
```



```
   // sets up animation control structure using this class
   public static void main(String[] args) {
      SimulationControl.createApp(new BouncingBallApp());
   }
}
```

Because we will advance the dynamical variables of each ball using a loop, we store them in an *array*. An array such as `ball` is a data structure that holds many objects (or primitive data) of the same type. The elements of an array are accessed using an index in square brackets. The index begins at 0 and ends at the length of the array minus 1. Arrays are created with the `new` operator and have several properties such as `length`. We will discuss arrays in more detail in Section 3.4.

In Listing 2.13 we represent each ball as an object of type `BouncingBall` in an array. This use of objects is appealing, but for better performance, it is usually better to store the positions and the velocities of the balls in an array of doubles. In Chapter 8 we will simulate a system of N mutually interacting particles. Because computational speed will be very important in this case, we will not allocate separate objects for each particle, and instead will treat the system of N particles as one object.

The `initialize` method in `BouncingBallApp` reads the number of particles and creates an array of the appropriate length. Creating an array sets primitive variables to zero and object values to null. For this reason we next loop to create the balls and add each ball to the frame. We place each ball initially at (0,0) with a random velocity. To produce random angles for the initial velocity, the `Math.random()` method is used. This method returns a random double between 0 and 1, not including the exact value 1. We define the random angle to be between 0 and π so that the initial vertical component of the velocity is positive. Clicking the Initialize button removes old objects from the drawing.

Most programming languages, including Java, use pixels to define the location on a window, with the origin at the upper left-hand corner and the vertical coordinate increasing in the downward direction. This choice of coordinates is usually not convenient in physics, and it often is more convenient to choose coordinates such that the vertical coordinate increases upward. The `Circle.setXY` method uses *world* or physical coordinates to set the position of the circle, and its implementation converts these coordinates to pixels so that the Java graphics methods can be used. In `initialize` we set the boundaries for the *world* coordinates using the `setPreferredMinMax` method whose arguments are the minimum x-coordinate, maximum x-coordinate, minimum y-coordinate, and maximum y-coordinate, respectively.

The `doStep` method implements a straightforward loop to advance the dynamical state of each ball in the array. It then advances the time and displays the time in the frame. Frames are automatically redrawn each time the `doStep` method is executed.

Finally, we note that there are two types of input parameters. Some parameters, such as the number of particles, determine properties of the model that should not be changed after the model has been created. We refer to these parameters as *fixed* because their values should be determined when the model is initialized. Other parameters, such as the time step Δt, can be changed between computations, but should not be changed during a computation. For example, if the time step is changed while a differential equation is being solved, one variable might be advanced using the old value of the time step while another variable is advanced using the new value. This type of synchronization error can be avoided by reading