(We have written Δx^2 rather than $\langle \Delta x^2 \rangle$ for simplicity.) To determine Δx^2, we write (7.8a) as the sum of two terms:

$$\Delta x^2 = \left\langle \sum_{i=1}^{N} \Delta_i \sum_{j=1}^{N} \Delta_j \right\rangle = \left\langle \sum_{i=1}^{N} \Delta_i^2 \right\rangle + \left\langle \sum_{i \neq j=1}^{N} \Delta_i \Delta_j \right\rangle, \tag{7.9}$$

where $\Delta_i = s_i - \langle s \rangle$. The first sum on the right-hand side of (7.9) includes terms for which $i = j$; the second sum is over i and j with $i \neq j$. Because each step is independent, we have $\langle \Delta_i \Delta_j \rangle = \langle \Delta_i \rangle \langle \Delta_j \rangle$. This term is zero because $\langle \Delta_i \rangle = 0$ for any i. The first term in (7.9) equals $N\langle \Delta^2 \rangle = N[\langle s^2 \rangle - \langle s \rangle^2] = N[a^2 - (p-q)^2 a^2] = N4pqa^2$, where we have used the fact that $p + q = 1$. Hence,

$$\Delta x^2 = 4pqNa^2 \quad \text{(analytical result)}. \tag{7.10}$$

We can gain more insight into the nature of random walks by doing a Monte Carlo simulation, that is, by using a computer to "flip coins." The implementation of the random walk algorithm is simple, for example,

```
if (p < Math.random()) {
   x++;
}
else {
   x--;
}
```

Clearly we have to sample many N step walks because, in general, each walk will give a different outcome. We need to do a Monte Carlo simulation many times and average over the results to obtain meaningful averages. Each N-step walk is called a *trial*. How do we know how many trials to use? The simple answer is to average over more and more trials until the average results don't change within the desired accuracy. The more sophisticated answer is to do an error analysis similar to what we do in measurements in the laboratory. Such an analysis is discussed in Section 11.4.

The more difficult parts of a program to simulate random walks are associated with bookkeeping. The walker takes a total of N steps in each trial, and the net displacement x is computed after every step. Our convention will be to use a variable name ending in `Accumulator` (or `Accum`) to denote a variable that accumulates the value of some variable. In Listing 7.3 we provide two classes to be used to simulate random walks.

Listing 7.3 Listing of `Walker` class.

```
package org.opensourcephysics.sip.ch07;
public class Walker {
   // accumulated data on displacement of walkers, index is time
   int xAccum[], xSquaredAccum[];
   int N;         // maximum number of steps
   double p;      // probability of step to the right
   int position; // position of walker

   public void initialize() {
      xAccum = new int[N+1];
      xSquaredAccum = new int[N+1];
   }

   public void step() {
      position = 0;
      for(int t = 0;t<N;t++) {
         if(Math.random()<p) {
            position++;
         } else {
            position--;
         }
         // determine displacement of walker after each step
         xAccum[t+1] += position;
         xSquaredAccum[t+1] += position*position;
      }
   }
}
```


Listing 7.4 Target class for random walk simulation.

```
package org.opensourcephysics.sip.ch07;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class WalkerApp extends AbstractSimulation {
   Walker walker = new Walker();
   PlotFrame plotFrame =
        new PlotFrame("time", "<x>,<x^2>", "Averages");
   HistogramFrame distribution =
        new HistogramFrame("x", "H(x)", "Histogram");
   int trials; // number of trials

   public WalkerApp() {
      plotFrame.setXYColumnNames(0, "t", "<x>");
      plotFrame.setXYColumnNames(1, "t", "<x^2>");
   }

   public void initialize() {
      walker.p = control.getDouble("Probability p of step to right");
      walker.N = control.getInt("Number of steps N");
      walker.initialize();
      trials = 0;
   }

   public void doStep() {
      trials++;
      walker.step();
      distribution.append(walker.position);
      distribution.setMessage("trials = "+trials);
   }

   public void stopRunning() {
      plotFrame.clearData();
      for(int t = 0; t<=walker.N; t++) {
         double xbar = walker.xAccum[t]*1.0/trials;
         double x2bar = walker.xSquaredAccum[t]*1.0/trials;
         plotFrame.append(0, 1.0*t, xbar);
         plotFrame.append(1, 1.0*t, x2bar - xbar*xbar);
      }
      plotFrame.repaint();
   }
```