to $\langle E_d \rangle$ by

$$kT/J = \frac{4}{\ln\left(1 + 4J/\langle E_d \rangle\right)}. \tag{15.10}$$

The result (15.10) comes from replacing the integrals in (15.7) by sums over the possible demon energies. Note that in the limit $|J/E_d| \ll 1$, (15.10) reduces to $kT = E_d$ as expected.

The `IsingDemon` class implements the Ising model in one dimension using periodic boundary conditions and the demon algorithm. Once the initial configuration is chosen, the demon algorithm is similar to that described in Section 15.3. However, the spins in the one-dimensional Ising model must be chosen at random. As usual, we will choose units such that $J = 1$.

Listing 15.3 The implementation of the demon algorithm for the one-dimensional Ising model.

```
package org.opensourcephysics.sip.ch15;
import java.awt.*;
import org.opensourcephysics.frames.*;

public class IsingDemon {
   public int[] demonEnergyDistribution;
   int N;                // number of spins
   public int systemEnergy;
   public int demonEnergy = 0;
   public int mcs = 0; // number of MC steps per spin
   public double systemEnergyAccumulator = 0;
   public double demonEnergyAccumulator = 0;
   public int magnetization = 0;
   public double mAccumulator = 0, m2Accumulator = 0;
   public int acceptedMoves = 0;
   private LatticeFrame lattice;

   public IsingDemon(LatticeFrame displayFrame) {
      lattice = displayFrame;
   }

   public void initialize(int N) {
      this.N = N;
      lattice.resizeLattice(N, 1); // set lattice size
      lattice.setIndexedColor(1, Color.red);
      lattice.setIndexedColor(-1, Color.green);
      demonEnergyDistribution = new int[N];
      for(int i = 0;i<N;++i) {
         // all spins up, second argument is always 0 for 1D lattice
         lattice.setValue(i, 0, 1);
      }
      int tries = 0;
      int E = -N; // start system in ground state
      magnetization = N; // all spins up
      // try up to 10N attempted flips so that system has desired
      // energy
      while((E<systemEnergy)&&(tries<10*N)) {
         int k = (int) (N*Math.random());
         int dE = 2*lattice.getValue(k, 0)*
               (lattice.getValue((k+1)%N, 0)
               +lattice.getValue((k-1+N)%N, 0));
         if(dE>0) {
            E += dE;
            int newSpin = -lattice.getValue(k, 0);
            lattice.setValue(k, 0, newSpin);
            magnetization += 2*newSpin;
         }
         tries++;
      }
      systemEnergy = E;
      resetData();
   }

   public double temperature() {
      return 4.0/Math.log(1.0+4.0/(demonEnergyAccumulator/(mcs*N)));
   }

   public void resetData() {
      mcs = 0;
      systemEnergyAccumulator = 0;
      demonEnergyAccumulator = 0;
      mAccumulator = 0;
      m2Accumulator = 0;
      acceptedMoves = 0;
   }

   public void doOneMCStep() {
      for(int j = 0;j<N;++j) {
         int i = (int) (N*Math.random());
         int dE = 2*lattice.getValue(i, 0)*
               (lattice.getValue((i+1)%N, 0)
               +lattice.getValue((i-1+N)%N, 0));
         if(dE<=demonEnergy) {
            int newSpin = -lattice.getValue(i, 0);
            lattice.setValue(i, 0, newSpin);
            acceptedMoves++;
            systemEnergy += dE;
            demonEnergy -= dE;
            magnetization += 2*newSpin;
         }
         systemEnergyAccumulator += systemEnergy;
         demonEnergyAccumulator += demonEnergy;
         mAccumulator += magnetization;
         m2Accumulator += magnetization*magnetization;
         demonEnergyDistribution[demonEnergy]++;
      }
      mcs++;
   }
}
```

Note that for $B = 0$, the change in energy due to a spin flip is either 0 or $\pm 4J$. Hence, it is convenient to choose the initial energy of the system plus the demon to be an integer multiple of $4J$. Because the spins interact, it is difficult to choose an initial configuration of spins with precisely the desired energy. The procedure followed in method `initialize`