```
            return s%L==L-1 ? EMPTY : s+1; // right
         case 2:
            return s/L==0 ? EMPTY : s-L;   // down
         case 3:
            return s/L==L-1 ? EMPTY : s+L; // above
         default:
            return EMPTY;
         }
   }

   // fills order[] array with random permutation of site indices.
   // First order[] is set to the identity permutation. Then for
   // values of i in {1...N-1}, swap values of order[i] with
   // order[r], where r is a random index in {i+1...N}.
   private void setOccupationOrder() {
      for(int s = 0;s<N;s++) {
         order[s] = s;
      }
      for(int s = 0;s<N-1;s++) {
         int r = s+(int) (Math.random()*(N-s));
         int temp = order[s];
         order[s] = order[r];
         order[r] = temp;
      }
   }

   // utility method to square an integer
   private int sqr(int x) {
      return x*x;
   }

   // merges two root sites into one to represent cluster merging.
   // Use heuristic that the root of the smaller cluster points to
   // the root of the larger cluster to improve performance.
   // parent[root] stores negative cluster size.
   private int mergeRoots(int r1, int r2) {
      // clusters are uniquely identified by their root sites. If they
      // are the same, then the clusters are already merged, and we
      // need do nothing
      if(r1==r2) {
         return r1;
         // if r1 has smaller cluster size than r2,
         // reverse (r1,r2) labels
      } else if(-parent[r1]<-parent[r2]) {
         return mergeRoots(r2, r1);
      } else { // (-parent[r1] > -parent[r2])
         // update cluster count and second cluster moment to account
         // for loss of two small clusters and gain of
         // one bigger cluster
         numClusters[-parent[r1]]--;
         numClusters[-parent[r2]]--;
         numClusters[-parent[r1]-parent[r2]]++;
         secondClusterMoment += sqr(parent[r1]+parent[r2])
              -sqr(parent[r1]) -sqr(parent[r2]);
         // cluster at r1 now includes sites of old cluster at r2
         parent[r1] += parent[r2];
```



```
         // make r1 new parent of r2
         parent[r2] = r1;
         // if r2 touched left or right, then so does merged cluster r1
         touchesLeft[r1] |= touchesLeft[r2];
         touchesRght[r1] |= touchesRght[r2];
         // if cluster at r1 spans lattice, then remember its size
         if(touchesLeft[r1]&&touchesRght[r1]) {
            spanningClusterSize = -parent[r1];
         }
         // return new root site r1
         return r1;
      }
   }
}
```

Listing 12.3 Target class for clusters.

```
package org.opensourcephysics.sip.ch12;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class ClustersApp extends AbstractSimulation {
   Scalar2DFrame grid =
        new Scalar2DFrame("Newman-Ziff cluster algorithm");
   PlotFrame plot1 = new PlotFrame("p", "Mean cluster size",
        "Mean cluster size");
   PlotFrame plot2 = new PlotFrame("p", "P_infty", "P_infty");
   PlotFrame plot3 = new PlotFrame("p", "P_span", "P_span");
   PlotFrame plot4 = new PlotFrame("s", "<n_s>",
        "Cluster size distribution");
   Clusters lattice;
   double pDisplay;
   double[] meanClusterSize;
   double[] P_infinity;
   double[] P_span;           // probability of a spanning cluster
   double[] numClustersAccum; // number of clusters of size s
   int numberOfTrials;

   public void initialize() {
      int L = control.getInt("Lattice size L");
      grid.resizeGrid(L, L);
      lattice = new Clusters(L);
      pDisplay = control.getDouble("Display lattice at this p");
      grid.setMessage("p = "+pDisplay);
      plot4.setMessage("p = "+pDisplay);
      plot4.setLogScale(true, true);
      meanClusterSize = new double[L*L];
      P_infinity = new double[L*L];
      P_span = new double[L*L];
      numClustersAccum = new double[L*L+1];
      numberOfTrials = 0;
   }

   public void doStep() {
      control.clearMessages();
      control.println("Trial "+numberOfTrials);
      // adds sites to new cluster and accumulate results
```