```
      automaton.setValue(L/2, 0, 1);
      // choose color of empty and occupied sites
      automaton.setIndexedColor(0, java.awt.Color.YELLOW); // empty
      automaton.setIndexedColor(1, java.awt.Color.BLUE);   // occupied
      setRule(control.getInt("Rule number"));
      for(int t = 1;t<tmax;t++) {
         iterate(t, L);
      }
   }

   public void iterate(int t, int L) {
      for(int i = 0;i<L;i++) {
         // read the neighborhood bits around index i
         // using periodic boundary conditions
         int left = automaton.getValue((i-1+L)%L, t-1);
         int center = automaton.getValue(i, t-1);
         int right = automaton.getValue((i+1)%L, t-1);
         // encode left, center, and right bits into one
         // integer value between 0 and 7
         int neighborhood = (left<<2)+(center<<1)+(right<<0);
         // update[neighborhood] gives the new site value for
         // this neighborhood
         automaton.setValue(i, t, update[neighborhood]);
      }
   }

   public void setRule(int ruleNumber) {
      control.println("Rule = "+ruleNumber+"\n");
      control.println("111   110   101   100   011   010   001   000");
      for(int i = 7;i>=0;i--) {
         // (ruleNumber >>> i) shifts the contents of ruleNumber to the
         // right by i bits. In particular, the ith bit of ruleNumber
         // resides in the rightmost position of this expression. After
         // the bitwise "and" operation with the number 1, we are left
         // with either the number 0 or 1, depending on whether the ith
         // bit of ruleNumber was cleared or set.
         update[i] = ((ruleNumber>>>i)&1);
         control.print("  "+update[i]+"     ");
      }
      control.println();
   }

   public void reset() {
      control.setValue("Rule number", 90);
      control.setValue("Maximum time", 100);
      control.setValue("Linear dimension", 500);
   }

   public static void main(String args[]) {
      CalculationControl.createApp(new OneDimensionalAutomatonApp());
   }
}
```

The properties of all 256 one-dimensional cellular automata have been cataloged (see Wolfram, 1984). We explore some of the properties of one-dimensional cellular automata in Problems 14.1 and 14.3.

Problem 14.1 One-dimensional cellular automata

(a) What is the result of 13 & 12, 33 >> 1 (decimal representation) and 1101 & 0111 (binary representation)? Consider rule 90 and work out by hand the values of `update[]` according to method `setRule`.

(b) Use `OneDimensionalAutomatonApp` and consider rule 90 shown in Figure 14.1. This rule is also known as the modulo-two rule because the value of a site at step $t+1$ is the sum modulo 2 of its two neighbors at step t. Choose the initial configuration to be a single nonzero site (the seed) at the midpoint of the lattice. It is sufficient to consider the evolution for approximately twenty iterations. Is the resulting pattern of nonzero sites self-similar? If so, characterize the pattern by a fractal dimension.

(c) Determine the properties of a rule for which the value of a site at step $t+1$ is the sum modulo 2 of the values of its neighbors plus its own value at step t. This rule is equivalent to 10010110 or rule $150 = 2^1 + 2^2 + 2^4 + 2^7$. Start with a single seed site.

(d) Choose a random initial configuration for which the independent probability for each site to have the value 1 is $p = 1/2$; otherwise, the value of the site is 0. Determine the evolution of rule 90, rule 150, rule $18 = 2^1 + 2^4$ (00010010), rule $73 = 2^0 + 2^3 + 2^6$ (01001001), and rule 136 (10001000). How sensitive are the patterns that are formed to the initial conditions? Does the nature of the patterns depend on the use or nonuse of periodic boundary conditions? ■

Listing 14.2 A more efficient implementation of method `iterate` in `OneDimensionalAutomatonApp`.

```
public void iterate(int t, int L) {
   // encodes state(L-1) and state(0) in second and first bits of
   // neighborhood variable
   int neighborhood = (automaton.getValue(L-1,t-1)<<1) +
       automaton.getValue(0,t-1);
   for (int i = 0; i < L; i++) {
      // clear third bit of neighborhood, but keep second and first bits
      neighborhood = neighborhood & 3;
      // shift second and first bits of neighborhood to third and
      // second bits
      neighborhood = neighborhood << 1;
      // encode state(i+1) into first bit of neighborhood using
      // periodic boundary conditions
      neighborhood += automaton.getValue((i+1)%L, t-1);
      // neighborhood now encodes the three bits of state surrounding
      // index i at time t-1; with neighborhood as an index, the
      // update[] table gives the state at index i and time t.
      automaton.setValue(i,t,update[neighborhood]);
   }
}
```

Method `iterate` in class `OneDimensionalAutomatonApp` is not as efficient as possible because it does not use information about the neighborhood at site i to determine the neighborhood at site $i+1$. A more efficient implementation is given in Listing 14.2. To understand how this version of method `iterate` works, suppose that the lattice at $t = 0$ is 1011, and we want to determine the neighborhood of the site at $i = 0$. The answer is 6 in