```
 1:
 2: /**
 3:  *  The Field class defines an object that models a field full of foxes and
 4:  *  hounds. Descriptions of the methods you must implement appear below.
 5:  */
 6: public class Field
 7: {
 8:
 9:
10:    /**
11:     *  Creates an empty field of given width and height
12:     *
13:     *  @param width of the field.
14:     *  @param height of the field.
15:     */
16:    public Field (int width, int height)
17:    {
18:       _field = new FieldOccupant[width][height];
19:
20:    } // Field
21:
22:
23:    /**
24:     *  @return the width of the field.
25:     */
26:    public int getWidth()
27:    {
28:        return _field.length;
29:    } // getWidth
30:
31:
32:    /**
33:     *  @return the height of the field.
34:     */
35:    public int getHeight()
36:    {
37:       return _field[0].length;
38:    } // getHeight
39:
40:
41:    /**
42:     *  Place an occupant in cell (x, y).
43:     *
44:    *  @param x is the x-coordinate of the cell to place a mammal in.
45:    *  @param y is the y-coordinate of the cell to place a mammal in.
46:     *  @param toAdd is the occupant to place.
47:     */
48:    public void setOccupantAt(int x, int y, FieldOccupant toAdd)
49:    {
50:       _field[x][y] = toAdd;
51:
52:    } // setOccupantAt
53:
54:
55:    /**
56:    *  @param x is the x-coordinate of the cell whose contents are queried.
57:    *  @param y is the y-coordinate of the cell whose contents are queried.
58:     *
59:     *  @return occupant of the cell (or null if unoccupied)
60:     */
```

```
61:    public FieldOccupant getOccupantAt(int x, int y)
62:    {
63:       return _field[x][y];
64:    } // getOccupantAt
65:
```

```
66:
67:     /**
68:      * Corrects the x and y values in the two dimensional array to account for
69:      * the actual shape of the field and adjacent tiles
70:      *
71:      * @param val the value to be corrected
72:      * @param width whether or not it is dealing with an x value
73:      *
74:      * @return the corrected value for the coordinate
75:      */
76:     public int correctVal(int val, boolean width)
77:     {
78:        int correctedVal = val;
79:
80:        if (val != 0 && val * -1 == Math.abs(val))
81:        {
82:           if(width)
83:           {
84:              correctedVal = getWidth() + val;
85:           }
86:           else
87:           {
88:              correctedVal = getHeight() + val;
89:           }
90:        }
91:
92:        if(val > getWidth() - 1)
93:        {
94:           correctedVal =val - getWidth() + 1;
95:        }
96:        else if(!width && val > getHeight() - 1)
97:        {
98:           correctedVal = val - getHeight() + 1;
99:        }
100:       return correctedVal;
101:    }
102:
103:    /**
104:     *  Define any variables associated with a Field object here.  These
105:     *  variables MUST be private.
106:     */
107:    private FieldOccupant[][] _field;
108:
109: }
110:
```

```
1:
2: import java.awt.Color;
3:
4: /**
5:  * Abstract parent class for objects that can occupy a cell in the Field
6:  */
7: public abstract class FieldOccupant
8: {
9:    /**
10:    * @return the color to use for a cell containing a particular kind
11:    *         of occupant
12:    */
13:   abstract public Color getDisplayColor();
14: }
```

```
 1:
 2: import java.awt.Color;
 3:
 4: /**
 5:  * Foxes can display themselves
 6:  */
 7: public class Fox extends FieldOccupant
 8: {
 9:    /**
10:     * @return the color to use for a cell occupied by a Fox
11:     */
12:    @Override
13:    public Color getDisplayColor()
14:    {
15:       return Color.green;
16:    } // getDisplayColor
17:
18:
19:    /**
20:     * @return the text representation of a cell occupied by a fox
21:     * @override
22:     */
23:    public String toString()
24:    {
25:       return "F";
26:    }
27:
28: }
```

```
 1:
 2: import java.awt.Color;
 3:
 4: /**
 5:  * Hounds can display themselves
 6:  */
 7: public class Hound extends FieldOccupant
 8: {
 9:
10:    /**
11:     * @return the color to use for a cell occupied by a Hound
12:     */
13:    @Override
14:    public Color getDisplayColor()
15:    {
16:       return Color.red;
17:    } // getDisplayColor
18:
19:    /**
20:     * Makes "H" the representation for a hound in a cell and also show
21:     * the status of the hound in terms of how close it is to starving
22:     * @override
23:     */
24:    public String toString()
25:    {
26:       return "H" + _hasNotEaten;
27:    }
28:
29:
30:    /**
31:     * Tests to see if the hound has starved and if not it increments the number
32:     * of days that the hound has not eaten
33:     *
34:     * @return whether or not the hound has been starved to death
35:     */
36:    public boolean starving()
37:    {
38:       boolean starved = false;
39:
40:
41:       if(_hasNotEaten == DEFAULT_STARVE_TIME + 1)
42:       {
43:          starved = true;
44:       }
45:       else
46:       {
47:          _hasNotEaten++;
48:       }
49:
50:
51:       return starved;
52:
53:    }
54:
55:
56:
57:
58:
59:    // The default starve time for Hounds
60:    public static final int DEFAULT_STARVE_TIME = 3;
61:
```

```
1:
2: import java.awt.Color;
3:
4: /**
5:  * Hounds can display themselves
6:  */
7: public class Hound extends FieldOccupant
62:    public int _hasNotEaten;
63:
64: }
```

```
 1:
 2: import java.awt.*;
 3: import java.util.*;
 4:
 5: /**
 6:  *  The Simulation class is a program that runs and animates a simulation of
 7:  *  Foxes and Hounds.
 8:  *
 9:  *  @author 1828799
10:  */
11:
12: public class Simulation
13: {
14:
15:    // The constant CELL_SIZE determines the size of each cell on the screen
16:    // during animation.  (You may change this if you wish.)
17:    private static final int CELL_SIZE = 10;
18:    private static final String USAGE_MESSAGE = "Usage: java Simulation "
19:          + "[--graphics] [--width int] [--height int] [--starvetime int] "
20:          + "[--fox float] [--hound float]";
21:
22:
23:
24:    /**
25:     * Computes the next state of the field from the current state and
26:     * returns the new state
27:     *
28:     * @param currentState is the current state of the Field
29:     *
30:     * @return new field state after one timestep
31:     */
32:    private static Field performTimestep(Field currentState)
33:    {
34:       // Creates new field
35:       Field newField = new Field(currentState.getWidth(),
36:             currentState.getHeight());
37:
38:       // Counters
39:       int houndCount = 0;
40:       int foxCount = 0;
41:
42:       for (int i = 0; i < currentState.getWidth(); i++)
43:       {
44:          for(int j = 0; j < currentState.getHeight(); j++)
45:          {
46:             // Gets numbers of foxes and hounds in surrounding cells
47:             foxCount = countAnimal(i, j, true, currentState);
48:             houndCount = countAnimal(i, j, false, currentState);
49:
50:             if(currentState.getOccupantAt(i, j) instanceof Fox)
51:             {
52:                // Case 1: If the cell contains a fox
53:
54:                foxCount = foxCount - 1;
55:                if(houndCount > 1)
56:                {
57:                   newField.setOccupantAt(i, j, new Hound());
58:                }
59:                else if(houndCount == 1)
60:                {
61:                   newField.setOccupantAt(i, j, null);
```

```
 1:
 2: import java.awt.*;
 3: import java.util.*;
62:                 }
63:                 else if(houndCount == 0)
64:                 {
65:                    newField.setOccupantAt(i, j, new Fox());
```

```
66:                }
67:             }
68:             else if(currentState.getOccupantAt(i, j) instanceof Hound)

69:             {
70:                // Case 2: If the cell contains a hound
71:
72:                houndCount = houndCount - 1;
73:
74:
75:                if(((Hound) currentState.getOccupantAt(i, j)).starving())
76:                {
77:                   newField.setOccupantAt(i, j, null);
78:                }
79:                else if(foxCount == 0)
80:                {
81:                   newField.setOccupantAt(i, j,
82:                         ((Hound) currentState.getOccupantAt(i, j)));
83:                }
84:                else  if(foxCount > 0)
85:                {
86:                   newField.setOccupantAt(i, j, new Hound());
87:                }
88:
89:             }
90:             else
91:             {
92:                // Case 3: If the cell is empty
93:
94:                if(foxCount > 1 && houndCount < 2)
95:                {
96:                   newField.setOccupantAt(i, j, new Fox());
97:                }
98:                else if(foxCount > 1 && houndCount > 1)
99:                {
100:                  newField.setOccupantAt(i, j, new Hound());
101:               }
102:            }
103:         }
104:      }
105:
106:      return newField;
107:   } // performTimestep
108:
109:
110:   /**
111:    * Counts the foxes or hounds in the surrounding 8 tiles (also includes
112:    * the one in the middle)
113:    *
114:    * @param x the x-variable
115:    * @param y the y-variable
116:    * @param fox the boolean for whether or not it's counting foxes
117:    * @param currentState the current state of the field
118:    *
119:    * @return the number of foxes or hounds counted
120:    */
121:   public static int countAnimal(int x, int y, boolean fox, Field currentState)
122:   {
123:      int count = 0;
124:      int placeOne;
125:      int placeTwo;
126:
```

```
   66:                }
   67:             }
   68:             else if(currentState.getOccupantAt(i, j) instanceof Hound)
  127:       for(int i = x - 1; i < x + 2; i++)
  128:       {
  129:          for(int j = y - 1; j < y + 2; j++)
  130:          {
```

```
131:             // Corrects x and y variables
132:             placeOne = currentState.correctVal(i, true);
133:             placeTwo = currentState.correctVal(j, false);
134:
135:             if(fox && currentState.getOccupantAt(placeOne, placeTwo)
136:                   instanceof Fox)
137:             {
138:                   count++;
139:             }
140:             else if(!fox && currentState.getOccupantAt(placeOne, placeTwo)
141:                   instanceof Hound)
142:             {
143:                   count++;
144:             }
145:
146:          }
147:       }

148:       return count;
149:    }
150:
151:
152:
153:
154:    /**
155:     * Draws the current state of the field
156:     *
157:     * @param graphicsContext is an optional GUI window to draw to
158:     * @param theField is the object to display
159:     */
160:    private static void drawField(Graphics graphicsContext, Field theField)
161:    {
162:       // If we have a graphics context then update the GUI, otherwise
163:      // output text-based display
164:      if (graphicsContext != null)
165:       {
166:          // Iterate over the cells and draw the thing in that cell
167:          for (int i = 0; i < theField.getHeight(); i++)
168:          {
169:             for (int j = 0; j < theField.getWidth(); j++)
170:             {
171:                // Get the color of the object in that cell and set the
172:                //cell color
173:                if (theField.getOccupantAt(j,i) != null)
174:                {
175:                   graphicsContext.setColor(
176:                         theField.getOccupantAt(j,i).getDisplayColor());
177:                }
178:                else // Empty cells are white
179:                {
180:                   graphicsContext.setColor(Color.white);
181:                }
182:                graphicsContext.fillRect(j * CELL_SIZE,
183:                                         i * CELL_SIZE, CELL_SIZE, CELL_SIZE);
184:             } // for
185:          } // for
186:       }
187:       else // No graphics, just text
188:       {
189:          // Draw a line above the field
190:          for (int i = 0; i < theField.getWidth() * 2 + 1; i++)
191:          {
```

```
131:             // Corrects x and y variables
132:             placeOne = currentState.correctVal(i, true);
133:             placeTwo = currentState.correctVal(j, false);
192:             System.out.print("-");
193:          }
194:          System.out.println();
195:          // For each cell, display the thing in that cell
```

```
196:          for (int i = 0; i < theField.getHeight(); i++)
197:          {
198:             System.out.print("|"); // separate cells with '|'
199:             for (int j = 0; j < theField.getWidth(); j++)
200:             {
201:                if (theField.getOccupantAt(j,i) != null)
202:                {
203:                   System.out.print(theField.getOccupantAt(j,i)+"|");
204:                }
205:                else
206:                {
207:                   System.out.print(" |");
208:                }
209:             }
210:             System.out.println();
211:          } // for
212:
213:          // Draw a line below the field
214:          for (int i = 0; i < theField.getWidth() * 2 + 1; i++)
215:          {
216:             System.out.print("-");
217:          }
218:          System.out.println();
219:
220:       } // else
221:    } // drawField
222:
223:
224:    /**
225:     *  Main reads the parameters and performs the simulation and animation.
226:     */
227:    public static void main(String[] args) throws InterruptedException
228:    {
229:       /**
230:        *  Default parameters.  (You may change these if you wish.)
231:        */
232:       int width = 50;                                  // Default width
233:       int height  = 25;                                // Default height
234:       int starveTime = Hound.DEFAULT_STARVE_TIME;  // Default starvation time
235:       double probabilityFox = 0.5;                     // Default prob of fox
236:       double probabilityHound = 0.15;                  // Default prob of hound
237:       boolean graphicsMode = false;
238:       Random randomGenerator = new Random();
239:       Field theField = null;
240:
241:       // If we attach a GUI to this program, these objects will hold
242:       // references to the GUI elements
243:       Frame windowFrame = null;
244:       Graphics graphicsContext = null;
245:       Canvas drawingCanvas = null;
246:
247:       /*
248:        *  Process the input parameters. Switches we understand include:
249:       *   --graphics for "graphics" mode
250:       *   --width 999 to set the "width"
251:        *  --height 999 to set the height
252:        *  --starvetime 999 to set the "starve time"
253:        *  --fox 0.999 to set the "fox probability"
254:        *  --hound 0.999 to set the "hound probability"
255:        */
256:       for (int argNum=0; argNum < args.length; argNum++)
```

```
196:          for (int i = 0; i < theField.getHeight(); i++)
197:          {
198:             System.out.print("|"); // separate cells with '|'
257:       {
258:          try
259:          {
260:             switch(args[argNum])
```

```
261:             {
262:                case "--graphics":  // Graphics mode
263:                   graphicsMode = true;
264:                   break;
265:
266:                case "--width": // Set width
267:                   width = Integer.parseInt(args[++argNum]);
268:                   break;
269:
270:                case "--height": // set height
271:                   height = Integer.parseInt(args[++argNum]);
272:                   break;
273:
274:                case "--starvetime": // set 'starve time'
275:                   starveTime = Integer.parseInt(args[++argNum]);
276:                   break;
277:
278:                case "--fox": // set the probability for adding a fox
279:                   probabilityFox = Double.parseDouble(args[++argNum]);
280:                   break;
281:
282:                case "--hound": // set the probability for adding a hound
283:                   probabilityHound = Double.parseDouble(args[++argNum]);
284:                   break;
285:
286:                default: // Anything else is an error and we'll quit
287:                   System.err.println("Unrecognized switch.");
288:                   System.err.println(USAGE_MESSAGE);
289:                   System.exit(1);
290:             } // switch
291:          }
292:          catch (NumberFormatException | ArrayIndexOutOfBoundsException e)
293:          {
294:             System.err.println("Illegal or missing argument.");
295:             System.err.println(USAGE_MESSAGE);
296:             System.exit(1);
297:          }
298:       } // for
299:
300:       // Create the initial Field.
301:       theField = new Field(width, height);
302:
303:       // Visit each cell; randomly placing a Fox, Hound, or nothing in each.
304:       for (int i = 0; i < theField.getWidth(); i++)
305:       {
306:          for (int j = 0; j < theField.getHeight(); j++)
307:          {
308:             // If a random number is greater than or equal to the probability
309:             // of adding a fox, then place a fox
310:             if (randomGenerator.nextFloat() <= probabilityFox)
311:             {
312:                theField.setOccupantAt(i, j, new Fox());
313:             }
314:             // If a random number is less than or equal to the probability of
315:             // adding a hound, then place a hound. Note that if a fox
316:             // has already been placed, it remains and the hound is
317:             // ignored.
318:             if (randomGenerator.nextFloat() <= probabilityHound)
319:             {
320:                theField.setOccupantAt(i, j, new Hound());
321:             }
```

```
261:             {
262:                case "--graphics":  // Graphics mode
263:                   graphicsMode = true;
322:          } // for
323:       } // for
324:
325:       // If we're in graphics mode, then create the frame, canvas,
```

```
329:          windowFrame = new Frame("Foxes and Hounds");
330:          windowFrame.setSize(theField.getWidth() * CELL_SIZE +
10,
331:                              theField.getHeight() * CELL_SIZE +
30);
332:          windowFrame.setVisible(true);
333:
334:          // Create a "Canvas" we can draw upon; attach it to the
window.
335:          drawingCanvas = new Canvas();
336:          drawingCanvas.setBackground(Color.white);
337:          drawingCanvas.setSize(theField.getWidth()
* CELL_SIZE, 338:
              theField.getHeight() * CELL_SIZE); 339:
              windowFrame.add(drawingCanvas);
340:          graphicsContext =
drawingCanvas.getGraphics(); 341:      } //
if
342:
343:       // Loop infinitely, performing timesteps. We could
optionally stop
344:       // when the Field becomes empty or full, though there is no
345:       // guarantee either of those will ever arise...
346:       while (true)
347:       {
348:          Thread.sleep(1000);                          // Wait one
second
349:          drawField(graphicsContext, theField);        // Draw
the current state 350:
           theField = performTimestep(theField);          //
Simulate a timestep 351:                                  }
352:
353:    } // main
354:
355: }
```