```java

/**
 *  The Field class defines an object that models a field full of foxes and
 *  hounds. Descriptions of the methods you must implement appear below.
 */
public class Field
{


   /**
    *  Creates an empty field of given width and height
    *
    *  @param width of the field.
    *  @param height of the field.
    */
   public Field (int width, int height)
   {
      _field = new FieldOccupant[width][height];

   } // Field


   /**
    *  @return the width of the field.
    */
   public int getWidth()
   {
       return _field.length;
   } // getWidth


   /**
    *  @return the height of the field.
    */
   public int getHeight()
   {
      return _field[0].length;
   } // getHeight


   /**
    *  Place an occupant in cell (x, y).
    *
    *  @param x is the x-coordinate of the cell to place a mammal in.
    *  @param y is the y-coordinate of the cell to place a mammal in.
    *  @param toAdd is the occupant to place.
    */
   public void setOccupantAt(int x, int y, FieldOccupant toAdd)
   {
      _field[x][y] = toAdd;

   } // setOccupantAt


   /**
    *  @param x is the x-coordinate of the cell whose contents are queried.
    *  @param y is the y-coordinate of the cell whose contents are queried.
    *
    *  @return occupant of the cell (or null if unoccupied)
    */
   public FieldOccupant getOccupantAt(int x, int y)
   {
      return _field[x][y];
   } // getOccupantAt


   /**
    * Corrects the x and y values in the two dimensional array to account for
    * the actual shape of the field and adjacent tiles
    *
    * @param val the value to be corrected
    * @param width whether or not it is dealing with an x value
    *
    * @return the corrected value for the coordinate
    */
   public int correctVal(int val, boolean width)
   {
      int correctedVal = val;

      if (val != 0 && val * -1 == Math.abs(val))
      {
         if(width)
         {
            correctedVal = getWidth() + val;
         }
         else
         {
            correctedVal = getHeight() + val;
         }
      }

      if(val > getWidth() - 1)
      {
         correctedVal =val - getWidth() + 1;
      }
      else if(!width && val > getHeight() - 1)
      {
         correctedVal = val - getHeight() + 1;
      }
      return correctedVal;
   }

   /**
    *  Define any variables associated with a Field object here.  These
    *  variables MUST be private.
    */
   private FieldOccupant[][] _field;

}

```

```java

import java.awt.Color;

/**
 * Abstract parent class for objects that can occupy a cell in the Field
 */
public abstract class FieldOccupant
{
   /**
    * @return the color to use for a cell containing a particular kind
    *         of occupant
    */
   abstract public Color getDisplayColor();
}
```

```java

import java.awt.Color;

/**
 * Foxes can display themselves
 */
public class Fox extends FieldOccupant
{
   /**
    * @return the color to use for a cell occupied by a Fox
    */
   @Override
   public Color getDisplayColor()
   {
      return Color.green;
   } // getDisplayColor


   /**
    * @return the text representation of a cell occupied by a fox
    * @override
    */
   public String toString()
   {
      return "F";
   }

}
```

```

import java.awt.Color;

/**
 * Hounds can display themselves
 */
public class Hound extends FieldOccupant
{

   /**
    * @return the color to use for a cell occupied by a Hound
    */
   @Override
   public Color getDisplayColor()
   {
      return Color.red;
   } // getDisplayColor

   /**
    * Makes "H" the representation for a hound in a cell and also show
    * the status of the hound in terms of how close it is to starving
    * @override
    */
   public String toString()
   {
      return "H" + _hasNotEaten;
   }


   /**
    * Tests to see if the hound has starved and if not it increments the number
    * of days that the hound has not eaten
    *
    * @return whether or not the hound has been starved to death
    */
   public boolean starving()
   {
      boolean starved = false;


      if(_hasNotEaten == DEFAULT_STARVE_TIME + 1)
      {
         starved = true;
      }
      else
      {
         _hasNotEaten++;
      }


      return starved;

   }





   // The default starve time for Hounds
   public static final int DEFAULT_STARVE_TIME = 3;

   public int _hasNotEaten;

}
```

```java

import java.awt.*;
import java.util.*;

/**
 *  The Simulation class is a program that runs and animates a simulation of
 *  Foxes and Hounds.
 *
 *  @author 1828799
 */

public class Simulation
{

   // The constant CELL_SIZE determines the size of each cell on the screen
   // during animation.  (You may change this if you wish.)
   private static final int CELL_SIZE = 10;
   private static final String USAGE_MESSAGE = "Usage: java Simulation "
         + "[--graphics] [--width int] [--height int] [--starvetime int] "
         + "[--fox float] [--hound float]";



   /**
    * Computes the next state of the field from the current state and
    * returns the new state
    *
    * @param currentState is the current state of the Field
    *
    * @return new field state after one timestep
    */
   private static Field performTimestep(Field currentState)
   {
      // Creates new field
      Field newField = new Field(currentState.getWidth(),
            currentState.getHeight());

      // Counters
      int houndCount = 0;
      int foxCount = 0;

      for (int i = 0; i < currentState.getWidth(); i++)
      {
         for(int j = 0; j < currentState.getHeight(); j++)
         {
            // Gets numbers of foxes and hounds in surrounding cells
            foxCount = countAnimal(i, j, true, currentState);
            houndCount = countAnimal(i, j, false, currentState);

            if(currentState.getOccupantAt(i, j) instanceof Fox)
            {
               // Case 1: If the cell contains a fox

               foxCount = foxCount - 1;
               if(houndCount > 1)
               {
                  newField.setOccupantAt(i, j, new Hound());
               }
               else if(houndCount == 1)
               {
                  newField.setOccupantAt(i, j, null);
               }
               else if(houndCount == 0)
               {
                  newField.setOccupantAt(i, j, new Fox());
               }
            }
            else if(currentState.getOccupantAt(i, j) instanceof Hound)
            {
               // Case 2: If the cell contains a hound

               houndCount = houndCount - 1;


               if(((Hound) currentState.getOccupantAt(i, j)).starving())
               {
                  newField.setOccupantAt(i, j, null);
               }
               else if(foxCount == 0)
               {
                  newField.setOccupantAt(i, j,
                        ((Hound) currentState.getOccupantAt(i, j)));
               }
               else  if(foxCount > 0)
               {
                  newField.setOccupantAt(i, j, new Hound());
               }

            }
            else
            {
               // Case 3: If the cell is empty

               if(foxCount > 1 && houndCount < 2)
               {
                  newField.setOccupantAt(i, j, new Fox());
               }
               else if(foxCount > 1 && houndCount > 1)
               {
                  newField.setOccupantAt(i, j, new Hound());
               }
            }
         }
      }

      return newField;
   } // performTimestep


   /**
    * Counts the foxes or hounds in the surrounding 8 tiles (also includes
    * the one in the middle)
    *
    * @param x the x-variable
    * @param y the y-variable
    * @param fox the boolean for whether or not it’s counting foxes
    * @param currentState the current state of the field
    *
    * @return the number of foxes or hounds counted
    */
   public static int countAnimal(int x, int y, boolean fox, Field currentState)
   {
      int count = 0;
      int placeOne;
      int placeTwo;

      for(int i = x - 1; i < x + 2; i++)
      {
         for(int j = y - 1; j < y + 2; j++)
         {
            // Corrects x and y variables
            placeOne = currentState.correctVal(i, true);
            placeTwo = currentState.correctVal(j, false);

            if(fox && currentState.getOccupantAt(placeOne, placeTwo)
                  instanceof Fox)
            {
                  count++;
            }
            else if(!fox && currentState.getOccupantAt(placeOne, placeTwo)
                  instanceof Hound)
            {
                  count++;
            }

         }
      }
      return count;
   }




   /**
    * Draws the current state of the field
    *
    * @param graphicsContext is an optional GUI window to draw to
    * @param theField is the object to display
    */
   private static void drawField(Graphics graphicsContext, Field theField)
   {
      // If we have a graphics context then update the GUI, otherwise
      // output text-based display
      if (graphicsContext != null)
      {
         // Iterate over the cells and draw the thing in that cell
         for (int i = 0; i < theField.getHeight(); i++)
         {
            for (int j = 0; j < theField.getWidth(); j++)
            {
               // Get the color of the object in that cell and set the
               //cell color
               if (theField.getOccupantAt(j,i) != null)
               {
                  graphicsContext.setColor(
                        theField.getOccupantAt(j,i).getDisplayColor());
               }
               else // Empty cells are white
               {
                  graphicsContext.setColor(Color.white);
               }
               graphicsContext.fillRect(j * CELL_SIZE,
                                        i * CELL_SIZE, CELL_SIZE, CELL_SIZE);
            } // for
         } // for
      }
      else // No graphics, just text
      {
         // Draw a line above the field
         for (int i = 0; i < theField.getWidth() * 2 + 1; i++)
         {
            System.out.print("-");
         }
         System.out.println();
         // For each cell, display the thing in that cell
         for (int i = 0; i < theField.getHeight(); i++)
         {
            System.out.print("|"); // separate cells with ’|’
            for (int j = 0; j < theField.getWidth(); j++)
            {
               if (theField.getOccupantAt(j,i) != null)
               {
                  System.out.print(theField.getOccupantAt(j,i)+"|");
               }
               else
               {
                  System.out.print(" |");
               }
            }
            System.out.println();
         } // for

         // Draw a line below the field
         for (int i = 0; i < theField.getWidth() * 2 + 1; i++)
         {
            System.out.print("-");
         }
         System.out.println();

      } // else
   } // drawField


   /**
    *  Main reads the parameters and performs the simulation and animation.
    */
   public static void main(String[] args) throws InterruptedException
   {
      /**
       *  Default parameters.  (You may change these if you wish.)
       */
      int width = 50;                              // Default width
      int height  = 25;                            // Default height
      int starveTime = Hound.DEFAULT_STARVE_TIME;  // Default starvation time
      double probabilityFox = 0.5;                 // Default prob of fox
      double probabilityHound = 0.15;              // Default prob of hound
      boolean graphicsMode = false;
      Random randomGenerator = new Random();
      Field theField = null;

      // If we attach a GUI to this program, these objects will hold
      // references to the GUI elements
      Frame windowFrame = null;
      Graphics graphicsContext = null;
      Canvas drawingCanvas = null;

      /*
       *  Process the input parameters. Switches we understand include:
       *  --graphics for "graphics" mode
       *  --width 999 to set the "width"
       *  --height 999 to set the height
       *  --starvetime 999 to set the "starve time"
       *  --fox 0.999 to set the "fox probability"
       *  --hound 0.999 to set the "hound probability"
       */
      for (int argNum=0; argNum < args.length; argNum++)
      {
         try
         {
            switch(args[argNum])
            {
               case "--graphics":  // Graphics mode
                  graphicsMode = true;
                  break;

               case "--width": // Set width
                  width = Integer.parseInt(args[++argNum]);
                  break;

               case "--height": // set height
                  height = Integer.parseInt(args[++argNum]);
                  break;

               case "--starvetime": // set 'starve time'
                  starveTime = Integer.parseInt(args[++argNum]);
                  break;

               case "--fox": // set the probability for adding a fox
                  probabilityFox = Double.parseDouble(args[++argNum]);
                  break;

               case "--hound": // set the probability for adding a hound
                  probabilityHound = Double.parseDouble(args[++argNum]);
                  break;

               default: // Anything else is an error and we'll quit
                  System.err.println("Unrecognized switch.");
                  System.err.println(USAGE_MESSAGE);
                  System.exit(1);
            } // switch
         }
         catch (NumberFormatException | ArrayIndexOutOfBoundsException e)
         {
            System.err.println("Illegal or missing argument.");
            System.err.println(USAGE_MESSAGE);
            System.exit(1);
         }
      } // for

      // Create the initial Field.
      theField = new Field(width, height);

      // Visit each cell; randomly placing a Fox, Hound, or nothing in each.
      for (int i = 0; i < theField.getWidth(); i++)
      {
         for (int j = 0; j < theField.getHeight(); j++)
         {
            // If a random number is greater than or equal to the probability
            // of adding a fox, then place a fox
            if (randomGenerator.nextFloat() <= probabilityFox)
            {
               theField.setOccupantAt(i, j, new Fox());
            }
            // If a random number is less than or equal to the probability of
            // adding a hound, then place a hound. Note that if a fox
            // has already been placed, it remains and the hound is
            // ignored.
            if (randomGenerator.nextFloat() <= probabilityHound)
            {
               theField.setOccupantAt(i, j, new Hound());
            }
         } // for
      } // for

      // If we're in graphics mode, then create the frame, canvas,
      // and window. If not in graphics mode, these will remain null
      if (graphicsMode)
      {
         windowFrame = new Frame("Foxes and Hounds");
         windowFrame.setSize(theField.getWidth() * CELL_SIZE + 10,
                             theField.getHeight() * CELL_SIZE + 30);
         windowFrame.setVisible(true);

         // Create a "Canvas" we can draw upon; attach it to the window.
         drawingCanvas = new Canvas();
         drawingCanvas.setBackground(Color.white);
         drawingCanvas.setSize(theField.getWidth() * CELL_SIZE,
                               theField.getHeight() * CELL_SIZE);
         windowFrame.add(drawingCanvas);
         graphicsContext = drawingCanvas.getGraphics();
      } // if

      // Loop infinitely, performing timesteps. We could optionally stop
      // when the Field becomes empty or full, though there is no
      // guarantee either of those will ever arise...
      while (true)
      {
         Thread.sleep(1000);                         // Wait one second
         drawField(graphicsContext, theField);       // Draw the current state
         theField = performTimestep(theField);       // Simulate a timestep
      }

   } // main

}
```