PPA Assignment 3

Author:

HoukangZhang k1924824

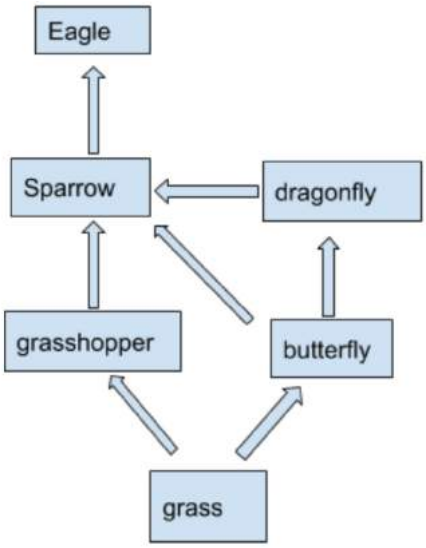
Yeshuai Cui k1924221

King's College London

Due Date: 2020/02/22

Introduction

This project simulates the food chain in a grassland, which consists of grass and 5 kinds of animals: eagle, sparrow, grasshopper, butterfly and dragonfly. All creatures may die because of age and eaten by other species. Animals may also die because of starvation and disease. The food chain is shown below:



Core Tasks

1. The relationship between different species are shown in the diagram and grass is implemented to grow and expand by reproduction, once a grass is created, it is not allowed to move.
2. **Sparrow** and **Dragonfly** both eats **Butterfly**. Both **Butterfly** and **Grasshopper** eat **Grass**.
3. All animals are designed to have genders, this is implemented by having a *protected* Boolean field in **Animal** Class called **isMale**

which is true for male and false for female. **getGender()** method will return this Boolean value. All the **giveBirth()** method for animals are called by female objects and if there is an male object around then they may have offspring. By calling **breed()** method, this will first determine whether there will be offsprings this step by check a random generated *double* value lower than **BREEDING_PROBABILITY** and random number of offspring are generated at maximum of **MAX_LITTER_SIZE**.

4. Time is implemented in the class **Condition**, **hours** is stored as an *double* value and each step is 15 minutes, 6 to 18 is daytime and night otherwise. Time is displayed at the bottom of window of simulator. **Eagle** may move 2 times at night with favored weather (see Challenge tasks 2.)

Challenge tasks

1. Plant is implemented by **Grass** class. Its growth rate is related to weather conditions. (see Challenge tasks 2.).
2. Weather is implemented in class **Condition**. Weather is updated each day and **getWeather()** is used to get an int value to represent weather, 0 for drought, 1 for normal and 2 for wet. This is also shown at the bottom of the simulator window. Animals

may act differently under different weathers. For example, butterfly may do nothing when weather is drought.

3. Disease is a feature that only animals have. Any animal has a possibility to become a source of outbreak of a disease and a possibility to cure itself. Disease is also spread between animals of same species when they are close to each other. An animal will die if has had a disease for certain steps and hasn't cured itself. This is implemented in method **illness()** in class **Animal** and is called by **act()** in **Animal**'s subclasses.

Run the program

Create a Simulator object. Then call one of:

simulateOneStep - for a single step.

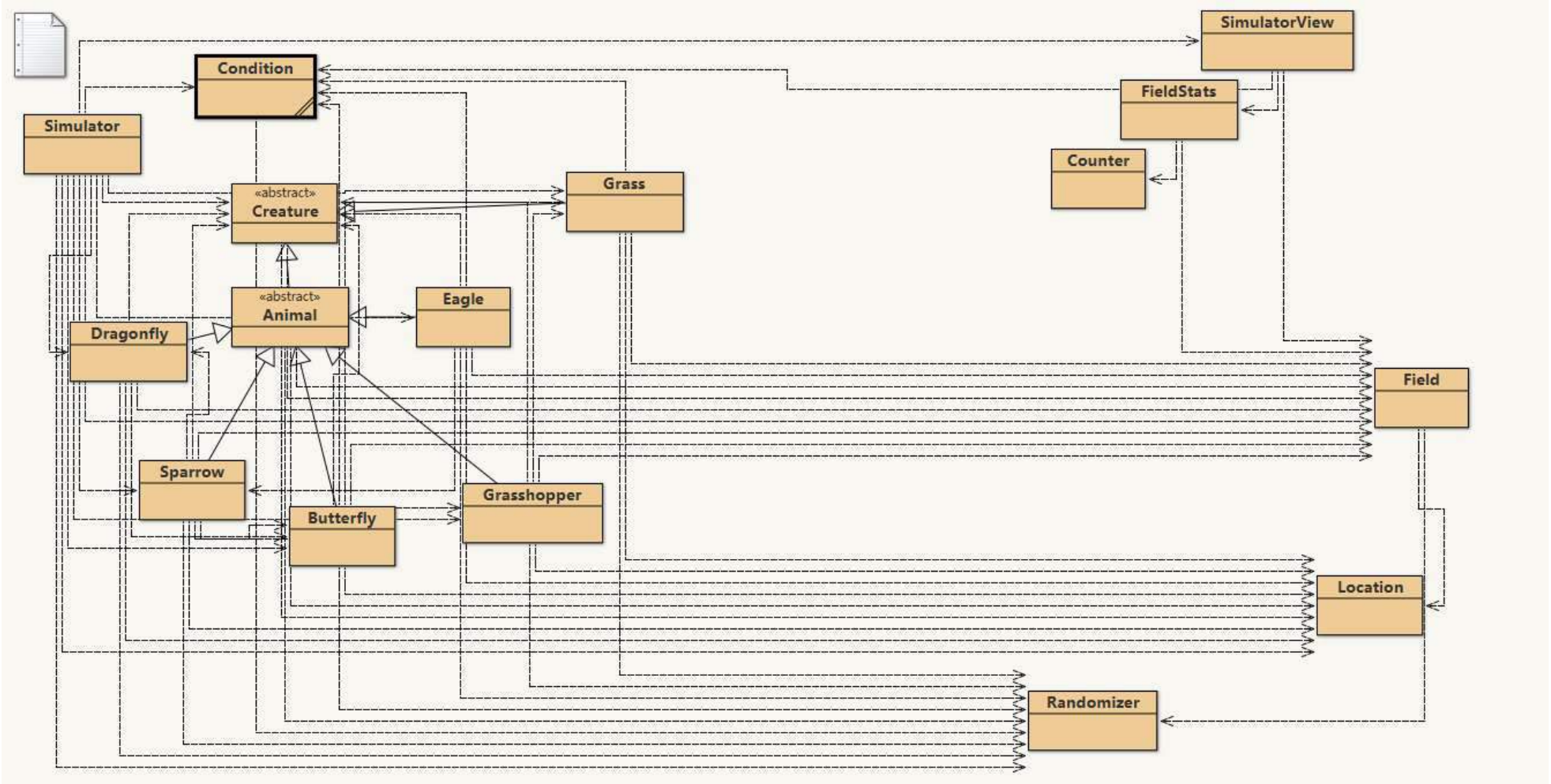
simulate - and supply a number (say 10) for that many steps.

runLongSimulation - for a simulation of 500 steps.

Tips: all the values are made for default size （250 * 250） of the simulator, there may be unexpected outcome with other self-defined size.

Limitation

Infected animals are unable to shown in the grid.


SimulatorView
Condition
FieldStats
Simulator
Counter
Grass
«abstract»
Creature
«abstract»
Animal
Eagle
Dragonfly
Field
Sparrow
Grasshopper
Butterfly
Location
Randomizer

```java
import java.util.Random;

/**
 * This class is used for conditions that will influence the acts
 * of creatures. Weather, Time
 * @author Houkang Zhang and Yeshuai Cui
 * @version 2020/02/20
 */
public class Condition
{
    // Current time is day or night
    private static boolean dayTime;
    // Hours of the day, initialize at morning
    private static double hours = 6;
    // Time elapses for one step
    private static double timePerStep = 0.25;
    // int value for weather. 0 for drought, 1 for normal, 2 for wet
    private static int weather = 1;
    // generate Random value for weather
    private static Random rand = Randomizer.getRandom();

    /**
     * update time and weather when this is called,
     * each update is 15 minutes.
     * weather changes each day
     */
    public static void update()
    {
        hours += timePerStep;
        if((hours >= 6) && (hours <= 18))
            dayTime = true;
        else
            dayTime = false;

        if(hours == 24)
        {
            hours = 0;
            double weatherDouble = rand.nextDouble();
            if(weatherDouble < 0.25)
                weather = 0;
            else if(weatherDouble > 0.75)
                weather = 2;
            else
                weather = 1;
        }

    }

    /**
     * @return int value represents weather condition
     */
    public static int getWeather()
    {
        return weather;
```

```
55      }
56
57      /**
58       * @return a boolean value, true for day and false for night
59       */
60      public static boolean getIfDay()
61      {
62          return dayTime;
63      }
64
65      /**
66       * @return an int value for hour of the day, 24 hours representation
67       */
68      public static int getHours()
69      {
70          return (int)hours;
71      }
72  }
73
```

```java
import java.util.Random;

/**
 * Provide control over the randomization of the simulation. By using the shared, fixed-seed
 * randomizer, repeated runs will perform exactly the same (which helps with testing). Set
 * 'useShared' to false to get different random behaviour every time.
 *
 * @author David J. Barnes and Michael Kölling
 * @version 2016.02.29
 */

public class Randomizer
{
    // The default seed for control of randomization.
    private static final int SEED = 1111;
    // A shared Random object, if required.
    private static final Random rand = new Random(SEED);
    // Determine whether a shared random generator is to be provided.
    private static final boolean useShared = true;

    /**
     * Constructor for objects of class Randomizer
     */
    public Randomizer()
    {
    }

    /**
     * Provide a random generator.
     * @return A random object.
     */
    public static Random getRandom()
    {
        if(useShared)
        {
            return rand;
        }
        else
        {
            return new Random();
        }
    }

    /**
     * Reset the randomization.
     * This will have no effect if randomization is not through
     * a shared Random generator.
     */
    public static void reset()
    {
        if(useShared)
        {
```

```
53                rand.setSeed(SEED);
54            }
55        }
56    }
57
```

```java
import java.util.List;
import java.util.Iterator;
import java.util.Random;
import javafx.util.Pair;

/**
 * A simple model of a Dragonfly.
 * Dragonflyes age, move, eat butterfly, it may infected by
 * disease and die because of disease and starvation. Condition will influence
 * the act of dragonfly
 *
 * @author David J. Barnes and Michael Kölling
 * @version 2016.02.29 (2)
 */
public class Dragonfly extends Animal
{
    // Characteristics shared by all Dragonflies (class variables).

    // The age at which a Dragonfly can start to breed.
    private static final int BREEDING_AGE = 5;
    // The age to which a Dragonfly can live.
    private static final int MAX_AGE = 40;
    // The likelihood of a Dragonfly breeding.
    private static final double BREEDING_PROBABILITY = 0.77;
    // The maximum number of births.
    private static final int MAX_LITTER_SIZE = 5;
    // The food value of a single butterfly. In effect, this is the
    // number of steps a Dragonfly can go before it has to eat again.
    private static final int BUTTERFLY_FOOD_VALUE = 35;
    // A shared random number generator to control breeding.
    private static final Random rand = Randomizer.getRandom();

    // Individual characteristics (instance fields).

    /**
     * Create a Dragonfly. A Dragonfly can be created as a new born (age zero
     * and not hungry) or with a random age and food level.
     *
     * @param randomAge If true, the Dragonfly will have random age and hunger
level.
     * @param field The field currently occupied.
     * @param location The location within the field.
     */
    public Dragonfly(boolean randomAge, Field field, Location location)
    {
        super(field, location);
        if(randomAge)
        {
            age = rand.nextInt(MAX_AGE);
            //the food value of new created dragonfly
            foodLevel = rand.nextInt(BUTTERFLY_FOOD_VALUE);
        }
        else
        {
```

```
            age = 0;
            //food value of new born dragonfly
            foodLevel = (int)(BUTTERFLY_FOOD_VALUE * 0.4);
        }
    }

    /**
     * This is what the Dragonfly does most of the time: it hunts for
     * butterfly and grasshopper. In the process, it might breed, die because
     * of starvation, age or disease.
     * @param field The field currently occupied.
     * @param newDragonflies A list to return newly born Dragonflies.
     */
    public void act(List<Creature> newDragonflies)
    {
        incrementAge();
        incrementHunger();
        illness();
        if(isAlive())
        {
            giveBirth(newDragonflies);
            // Move towards a source of food if found.
            Location newLocation = findFood();
            if(newLocation == null)
            {
                // No food found - try to move to a free location.
                newLocation = getField().freeAdjacentLocation(getLocation());
            }
            // See if it was possible to move.
            if(newLocation != null)
            {
                setLocation(newLocation);
            }
            else
            {
                // Overcrowding.
                setDead();
            }
        }
    }

    /**
     * Increase the age. This could result in the Dragonfly's death.
     */
    private void incrementAge()
    {
        age++;
        if(age > MAX_AGE)
        {
            setDead();
        }
    }

    /**
```

```
108      * Look for butterfly adjacent to the current location.
109      * Only the first found butterfly is eaten.
110      * @return Where food was found, or null if it wasn't.
111      */
112     private Location findFood()
113     {
114         Iterator<Location> it = adjacentIterator();
115         while(it.hasNext())
116         {
117             Location where = it.next();
118             Object animal = field.getObjectAt(where);
119             if(animal instanceof Butterfly)
120             {
121                 Butterfly butterfly = (Butterfly) animal;
122                 if(butterfly.isAlive())
123                 {
124                     butterfly.setDead();
125                     foodLevel = BUTTERFLY_FOOD_VALUE;
126                     return where;
127                 }
128             }
129         }
130         return null;
131     }
132
133     /**
134      * If dragonfly is ill it may spread disease to adjacent dragonflies
135      */
136     public void findInfect()
137     {
138         Iterator<Location> it = adjacentIterator();
139         while(it.hasNext())
140         {
141             Location where = it.next();
142             Object animal = field.getObjectAt(where);
143             if(rand.nextDouble() <= ILL_PROBABILITY &&isIll() && animal
    instanceof Dragonfly)
144             {
145                 Dragonfly dragonfly = (Dragonfly) animal;
146                 if(dragonfly.isAlive())
147                 {
148                     dragonfly.setIll();
149                 }
150             }
151         }
152     }
153
154     /**
155      * Check whether or not this Dragonfly is to give birth at this step.
156      * New births will be made into free adjacent locations.
157      * @param newDragonflies A list to return newly born Dragonflies.
158      */
159     private void giveBirth(List<Creature> newDragonfly)
160     {
```

```
161         // New Dragonflyes are born into adjacent locations.
162         // Get a list of adjacent free locations.
163         // this method is only called by female dragonfly
164         if(!getGender())
165         {
166             Field field = getField();
167             List<Location> adjacent = field.adjacentLocations(getLocation());
168             Iterator<Location> it = adjacent.iterator();
169             while(it.hasNext())
170             {
171                 Location where = it.next();
172                 Object what = field.getObjectAt(where);
173                 if(what instanceof Dragonfly)
174                 {
175                     Dragonfly dragonfly = (Dragonfly) what;
176                     //only reproduce when this two dragonfly are of opposite
    gender
177                     if(dragonfly.getGender())
178                     {
179                         List<Location> free =
    field.getFreeAdjacentLocations(getLocation());
180                         int births = breed();
181                         for(int b = 0; b < births && free.size() > 0; b++) {
182                             Location loc = free.remove(0);
183                             Dragonfly young = new Dragonfly(false, field, loc);
184                             newDragonfly.add(young);
185                             break;
186                         }
187                     }
188                 }
189             }
190         }
191     }
192
193     /**
194      * Generate a number representing the number of births,
195      * if it can breed.
196      * @return The number of births (may be zero).
197      */
198     private int breed()
199     {
200         int births = 0;
201         if(canBreed() && rand.nextDouble() <= BREEDING_PROBABILITY && foodLevel >
    8)
202         {
203             births = rand.nextInt(MAX_LITTER_SIZE) + 1;//inclusive
204         }
205         if(births > 0)
206         {
207             foodLevel -= 1 ;//dragonfly will lose foodlevel when give birth
208             //to its offsprings.
209         }
210         return births;
211     }
```

```
212
213     /**
214      * @return boolean
215      * A Dragonfly can breed if it has reached the breeding age.
216      */
217     private boolean canBreed()
218     {
219         return age >= BREEDING_AGE;
220     }
221 }
222
```

```java
import java.util.List;

/**
 * A class representing shared characteristics of all creatures.
 *
 * @author David J. Barnes,  Michael Kölling, Houkang Zhang and Yeshuai Cui
 * @version 2020/02/20
 */
public abstract class Creature
{
    // Whether the creature is alive or not.
    private boolean alive;
    // The creature's field.
    protected Field field;
    // The creature's position in the field.
    private Location location;
    //The age of each creature
    protected int age;

    /**
     * Create a new animal at location in field.
     *
     * @param field The field currently occupied.
     * @param location The location within the field.
     */
    public Creature(Field field, Location location)
    {
        alive = true;
        this.field = field;
        setLocation(location);
    }

    /**
     * Make this ceature act - that is: make it do
     * whatever it wants/needs to do.
     * @param newCreatures A list to receive newly born animals.
     */
    abstract public void act(List<Creature> newCreatures);

    /**
     * Check whether the animal is alive or not.
     * @return true if the animal is still alive.
     */
    protected boolean isAlive()
    {
        return alive;
    }

    /**
     * Indicate that the animal is no longer alive.
     * It is removed from the field.
     */
    protected void setDead()
    {
```

```
        alive = false;
        if(location != null)
        {
            field.clear(location);
            location = null;
            field = null;
        }
    }

    /**
     * Return the animal's location.
     * @return The animal's location.
     */
    protected Location getLocation()
    {
        return location;
    }

    /**
     * Place the animal at the new location in the given field.
     * @param newLocation The animal's new location.
     */
    protected void setLocation(Location newLocation)
    {
        if(location != null)
        {
            field.clear(location);
        }
        location = newLocation;
        field.place(this, newLocation);
    }

    /**
     * Return the animal's field.
     * @return The animal's field.
     */
    protected Field getField()
    {
        return field;
    }
}

```

```java
import java.util.*;
import javafx.util.Pair;

/**
 * A simple model of a Butterfly.
 * Butterflies age, move, eat butterfly, it may infected by
 * disease and die because of disease and starvation. Condition will influence
 * the act of butterfly
 * @author David J. Barnes,  Michael Kölling, Houkang Zhang and Yeshuai Cui
 * @version 2020/02/20
 */
public class Butterfly extends Animal
{
    // Characteristics shared by all butterflys (class variables).
    // The age at which a butterfly can start to breed.
    private static final int BREEDING_AGE = 4;
    // The age to which a butterfly can live.
    private static final int MAX_AGE = 30;
    // The likelihood of a butterfly breeding.
    private static final double BREEDING_PROBABILITY = 0.78;
    // The maximum number of births.
    private static final int MAX_LITTER_SIZE = 7;
    // The food value of a single butterfly. In effect, this is the
    // number of steps a butterfly can go before it has to eat again.
    private static final int GRASS_FOOD_VALUE = 25;
    // A shared random number generator to control breeding.
    private static final Random rand = Randomizer.getRandom();
    // The possibility that this butterfly will do nothing during this step
    private static final double REST_PROBABILITY = 0.3;

    /**
     * Create a butterfly. A butterfly can be created as a new born (age zero
     * and not hungry) or with a random age and food level.
     *
     * @param randomAge If true, the butterfly will have random age and hunger
level.
     * @param field The field currently occupied.
     * @param location The location within the field.
     */
    public Butterfly(boolean randomAge, Field field, Location location)
    {
        super(field, location);
        if(randomAge)
        {
            age = rand.nextInt(MAX_AGE);
            foodLevel = rand.nextInt(GRASS_FOOD_VALUE);
        }
        else
        {
            age = 0;
            foodLevel = (int)(GRASS_FOOD_VALUE * 0.7);
        }
    }

```

```
/**
 * This is what the butterfly does most of the time: it hunts for
 * grass. In the process, it might breed, die because
 * of starvation, age or disease.
 * @param field The field currently occupied.
 * @param newButterfly A list to return newly born butterfly.
 */
public void act(List<Creature> newBuffterfly)
{
    incrementAge();
    if(Condition.getWeather() == 0 && rand.nextDouble() < REST_PROBABILITY)
    {
        return;
    }
    else
    {
        incrementHunger();
        illness();
        if(isAlive())
        {
            giveBirth(newBuffterfly);
            // Move towards a source of food if found.
            Location newLocation = findFood();
            if(newLocation == null)
            {
                // No food found - try to move to a free location.
                newLocation = getField().freeAdjacentLocation(getLocation());
            }
                // See if it was possible to move.
            if(newLocation != null)
            {
                setLocation(newLocation);
            }
            else
            {
                // Overcrowding.
                setDead();
            }
        }
    }
}

/**
 * Increase the age. This could result in the butterfly's death.
 */
private void incrementAge()
{
    age++;
    if(age > MAX_AGE)
    {
        setDead();
    }
}
```

```
108     /**
109      * Look for grass adjacent to the current location.
110      * Only the first live grass is eaten.
111      * @return Where food was found, or null if it wasn't.
112      */
113     private Location findFood()
114     {
115         Iterator<Location> it = adjacentIterator();
116         while(it.hasNext())
117         {
118             Location where = it.next();
119             Object animal = field.getObjectAt(where);
120             if(animal instanceof Grass)
121             {
122                 Grass grass = (Grass) animal;
123                 if(grass.isAlive())
124                 {
125                     grass.setDead();
126                     foodLevel = GRASS_FOOD_VALUE;
127                     return where;
128                 }
129             }
130         }
131         return null;
132     }
133
134     /**
135      * If butterfly is ill it may spread disease to adjacent butterflies
136      */
137     public void findInfect()
138     {
139         Iterator<Location> it = adjacentIterator();
140         while(it.hasNext())
141         {
142             Location where = it.next();
143             Object animal = field.getObjectAt(where);
144             if( rand.nextDouble() <= ILL_PROBABILITY &&isIll() && animal
    instanceof Butterfly)
145             {
146                 Butterfly butterfly = (Butterfly) animal;
147                 if(butterfly.isAlive())
148                 {
149                     butterfly.setIll();
150                 }
151             }
152         }
153     }
154
155     /**
156      * Check whether or not this butterfly is to give birth at this step.
157      * New births will be made into free adjacent locations.
158      * @param newButterfly A list to return newly born Butterfly.
159      */
160     private void giveBirth(List<Creature> newButterfly)
```

```
    {
        // New Dragonflyes are born into adjacent locations.
        // Get a list of adjacent free locations.
        // This method is called by female butterfly
        if(!getGender())
        {
            Field field = getField();
            List<Location> adjacent = field.adjacentLocations(getLocation());
            Iterator<Location> it = adjacent.iterator();
            while(it.hasNext())
            {
                Location where = it.next();
                Object what = field.getObjectAt(where);
                if(what instanceof Butterfly)
                {
                    Butterfly butterfly = (Butterfly) what;
                    //only reproduce when two sparrow are of opposite gender
                    if(butterfly.getGender())
                    {
                        List<Location> free =
field.getFreeAdjacentLocations(getLocation());
                        int births = breed();
                        for(int b = 0; b < births && free.size() > 0; b++) {
                            Location loc = free.remove(0);
                            Butterfly young = new Butterfly(false, field, loc);
                            newButterfly.add(young);
                            break;
                        }
                    }
                }
            }
        }
    }

    /**
     * Generate a number representing the number of births,
     * if it can breed.
     * @return The number of births (may be zero).
     */
    private int breed()
    {
        int births = 0;
        if(canBreed() && rand.nextDouble() <= BREEDING_PROBABILITY &&
foodLevel>5)
        {
            births = rand.nextInt(MAX_LITTER_SIZE) + 1;//inclusive
        }
        if(births > 0)
        {
            foodLevel -= 1 ;
            //this butterfly will decrease foodlevel
            //when give birth to a new butterfly
        }
        return births;
```

```
213         }
214
215         /**
216          * A butterfly can breed if it has reached the breeding age.
217          */
218         private boolean canBreed()
219         {
220             return age >= BREEDING_AGE;
221         }
222 }
```

```java
import java.util.*;
import javafx.util.Pair;

/**
 * A simple model of a Grasshopper.
 * grasshoppers age, move, eat grass, it may infected by
 * disease and die because of disease and starvation. Condition will influence
 * the act of grasshopper
 * @author David J. Barnes,  Michael Kölling, Houkang Zhang and Yeshuai Cui
 * @version 2020/02/20
 */

public class Grasshopper extends Animal
{
    // Characteristics shared by all Grasshopperss (class variables).
    // The age at which a grasshopper can start to breed.
    private static final int BREEDING_AGE = 5;//newBreedingAge
    // The age to which a grasshopper can live.
    private static final int MAX_AGE = 30;
    // The likelihood of a grasshopper breeding.
    private static final double BREEDING_PROBABILITY = 0.75;
    // The maximum number of births.
    private static final int MAX_LITTER_SIZE = 6;
    // The food value of a single grasshopper. In effect, this is the
    // number of steps a grasshopper can go before it has to eat again.
    private static final int GRASS_FOOD_VALUE = 30;
    // A shared random number generator to control breeding.
    private static final Random rand = Randomizer.getRandom();

    /**
     * Create a grasshopper. A grasshopper can be created as a new born
     * (age zero and not hungry) or with a random age and food level.
     *
     * @param randomAge If true, the grasshopper will have random age and hunger
level.
     * @param field The field currently occupied.
     * @param location The location within the field.
     */
    public Grasshopper(boolean randomAge, Field field, Location location)
    {
        super(field, location);
        if(randomAge)
        {
            age = rand.nextInt(MAX_AGE);
            foodLevel = rand.nextInt(GRASS_FOOD_VALUE);
        }
        else
        {
            age = 0;
            foodLevel = (int)(GRASS_FOOD_VALUE * 0.8);
        }
    }

    /**
```

```
54      * This is what the grasshopper does most of the time: it hunts for
55      * grass. In the process, it might breed, die because
56      * of starvation, age or disease.
57      * @param field The field currently occupied.
58      * @param newGrasshopper A list to return newly born grasshoppers.
59      */
60     public void act(List<Creature> newGrasshoppers)
61     {
62         incrementAge();
63         incrementHunger();
64         illness();
65         if(isAlive())
66         {
67             giveBirth(newGrasshoppers);
68             // Move towards a source of food if found.
69             Location newLocation = findFood();
70             if(newLocation == null)
71             {
72                 // No food found - try to move to a free location.
73                 newLocation = getField().freeAdjacentLocation(getLocation());
74             }
75             // See if it was possible to move.
76             if(newLocation != null)
77             {
78                 setLocation(newLocation);
79             }
80             else
81             {
82                 // Overcrowding.
83                 setDead();
84             }
85         }
86     }
87
88     /**
89      * Age is incremented agter each step
90      */
91     private void incrementAge()
92     {
93         age++;
94         if(age > MAX_AGE)
95         {
96             setDead();
97         }
98     }
99
100    /**
101     * Look for grass adjacent to the current location.
102     * Only the first live prey is eaten.
103     * @return Where food was found, or null if it wasn't.
104     */
105    private Location findFood()
106    {
107        Iterator<Location> it = adjacentIterator();
```

```
        while(it.hasNext())
        {
            Location where = it.next();
            Object creature = field.getObjectAt(where);
            if(creature instanceof Grass)
            {
                Grass grass = (Grass) creature;
                if(grass.isAlive())
                {
                    grass.setDead();
                    foodLevel = GRASS_FOOD_VALUE;
                    return where;
                }
            }
        }
        return null;
    }

    /**
     * If grasshopper is ill it may spread disease to adjacent dragonflies
     */
    public void findInfect()
    {
        Iterator<Location> it = adjacentIterator();
        while(it.hasNext())
        {
            Location where = it.next();
            Object animal = field.getObjectAt(where);
            if( rand.nextDouble() <= ILL_PROBABILITY &&isIll() && animal
instanceof Grasshopper)
            {
                Grasshopper grasshopper = (Grasshopper) animal;
                if(grasshopper.isAlive())
                {
                    grasshopper.setIll();
                }
            }
        }
    }

    /**
     * Check whether or not this grasshopper is to give birth at this step.
     * New births will be made into free adjacent locations.
     * @param newGrasshopper A list to return newly born grasshoppers.
     */
    private void giveBirth(List<Creature> newGrasshopper)
    {
        // New Dragonflyes are born into adjacent locations.
        // Get a list of adjacent free locations.
        // This method is called by female fragonfly
        if(!getGender())
        {
            Field field = getField();
            List<Location> adjacent = field.adjacentLocations(getLocation());
```

```
161             Iterator<Location> it = adjacent.iterator();
162             while(it.hasNext())
163             {
164                 Location where = it.next();
165                 Object what = field.getObjectAt(where);
166                 if(what instanceof Grasshopper)
167                 {
168                     Grasshopper grasshopper = (Grasshopper) what;
169                     //only reproduce when two grasshopper are of opposite gender
170                     if(grasshopper.getGender())
171                     {
172                         List<Location> free =
    field.getFreeAdjacentLocations(getLocation());
173                         int births = breed();
174                         for(int b = 0; b < births && free.size() > 0; b++)
175                         {
176                             Location loc = free.remove(0);
177                             Grasshopper young = new Grasshopper(false, field,
    loc);
178 
179                             newGrasshopper.add(young);
180                             break;
181                         }
182                     }
183                 }
184             }
185         }
186 
187         /**
188          * Generate a number representing the number of births,
189          * if it can breed.
190          * @return The number of births (may be zero).
191          */
192         private int breed()
193         {
194             int births = 0;
195             if(canBreed() && rand.nextDouble() <= BREEDING_PROBABILITY && foodLevel >
    10)
196             {
197                 births = rand.nextInt(MAX_LITTER_SIZE) + 1;
198             }
199             if(births > 0)
200             {
201                 foodLevel -= 2 ;
202             }
203             return births;
204         }
205 
206         /**
207          * A grasshopper can breed if it has reached the breeding age.
208          */
209         private boolean canBreed()
210         {
211             return age >= BREEDING_AGE;
```

```
212     }
213 }
```

```java
import java.util.List;
import java.util.Iterator;
import java.util.Random;

/**
 * A simple model of grass. Grass will expand and be eaten by animals
 * Condition will influence the act of grass
 *
 * *@author David J. Barnes,  Michael Kölling, Houkang Zhang and Yeshuai Cui
 * @version 2020/02/20
 */
public class Grass extends Creature
{
    // The age at which a grass can start to breed.
    private static int BREEDING_AGE =30;
    // The age to which a grass can live.
    private static int MAX_AGE = 100;
    // The likelihood of a grass breeding.
    private static double BREEDING_PROBABILITY = 0.1;
    // The maximum number of births.
    private static int LITTER_SIZE = 5;
    // A shared random number generator to control breeding.
    private static final Random rand = Randomizer.getRandom();
    //actual maximum number of births considering conditions
    private static int new_LITTER_SIZE;
    //actual likelihood of breeding consdering conditions
    private static double new_BREDDING_PROBABILITY;

    /**
     * @param randomAge If true, the grass will have random age
     * @param field The field currently occupied.
     * @param location The location within the field.
     */
    public Grass(boolean randomAge ,Field field , Location location)
    {
        super(field , location);
        if(randomAge)
        {
            age = rand.nextInt(MAX_AGE);
        }
        else
        {
            age = 0;
        }
        }

    /**
     * This is what the grass does most of the time: it might breed,
     * die because of age
     * @param field The field currently occupied.
     * @param newGrass A list to return newly born grass.
     */
    public void act(List<Creature> creature)
    {
```

```
        incrementAge();
        //parameters are updated accourding to conditions
        simulateCondition();
        if(isAlive())
        {
            giveBirth(creature);
        }
    }

    /**
     * Check whether or not this grass is to give birth at this step.
     * New births will be made into free adjacent locations.
     * @param newGrass A list to return newly born grass.
     */
    private void giveBirth(List<Creature> newGrass)
    {
        Field field = getField();
        List<Location> free = field.getFreeAdjacentLocations(getLocation());
        int birth = Breed();
        for(int n = 0;free.size() > 0 &&  n < birth && n < free.size() ; n++)
        {
            Location loc = free.remove(0);
            Grass young = new Grass(false, field, loc);
            newGrass.add(young);
        }
    }

    /**
     * newVALUEs are updated for different conditions and ready for further use
     */
    private void simulateCondition()
    {
        int c = Condition.getWeather();
        if(c == 1)
        {
            new_LITTER_SIZE = LITTER_SIZE;
            new_BREDDING_PROBABILITY = BREEDING_PROBABILITY;
        }
        else if(c == 2)
        {
            new_LITTER_SIZE = LITTER_SIZE + 1;
            new_BREDDING_PROBABILITY = BREEDING_PROBABILITY *1.2;
        }
        else
        {
            new_LITTER_SIZE = LITTER_SIZE - 1;
            new_BREDDING_PROBABILITY = BREEDING_PROBABILITY * 0.8;
        }
    }

    /**
     * Generate a number representing the number of births,
     * if it can breed.
     * @return The number of births (may be zero).
```

```
109         */
110        private int Breed()
111        {
112            if(rand.nextDouble() <= new_BREDDING_PROBABILITY)
113            {
114                return rand.nextInt(new_LITTER_SIZE);
115            }
116            return 0;
117        }
118
119        /**
120         * age is incremented after each step
121         */
122        public void incrementAge()
123        {
124            age++;
125            if(age >= MAX_AGE)
126            {
127                setDead();
128            }
129        }
130
131    }
```

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.LinkedHashMap;
import java.util.Map;

/**
 * A graphical view of the simulation grid.
 * The view displays a colored rectangle for each location
 * representing its contents. It uses a default background color.
 * Colors for each type of species can be defined using the
 * setColor method.
 *
 * @author David J. Barnes,  Michael Kölling, Houkang Zhang and Yeshuai Cui
 * @version 2020/02/20
 */
public class SimulatorView extends JFrame
{
    // Colors used for empty locations.
    private static final Color EMPTY_COLOR = Color.white;

    // Color used for objects that have no defined color.
    private static final Color UNKNOWN_COLOR = Color.gray;

    private final String STEP_PREFIX = "Step: ";
    private final String POPULATION_PREFIX = "Population: ";
    private final String WEATHER_PREFIX = "Weather: ";
    private final String TIME_PREFIX = "Time: ";
    private JLabel stepLabel, downwardinfo, infoLabel, weatherLabel, timeLabel;
    private FieldView fieldView;

    // A map for storing colors for participants in the simulation
    private Map<Class, Color> colors;
    // A statistics object computing and storing simulation information
    private FieldStats stats;

    /**
     * Create a view of the given width and height.
     * @param height The simulation's height.
     * @param width  The simulation's width.
     */
    public SimulatorView(int height, int width)
    {
        stats = new FieldStats();
        colors = new LinkedHashMap<>();
        //new added weatherLabel and timeLabel is to represent weather
        //and time in the bottom of the window
        setTitle("Fox and Rabbit Simulation");
        stepLabel = new JLabel(STEP_PREFIX, JLabel.CENTER);
        infoLabel = new JLabel("  ", JLabel.CENTER);
        downwardinfo = new JLabel(POPULATION_PREFIX, JLabel.CENTER);
        weatherLabel = new JLabel(WEATHER_PREFIX, JLabel.CENTER);
        timeLabel = new JLabel(TIME_PREFIX, JLabel.CENTER);
        setLocation(100, 50);
```

```
55
56          fieldView = new FieldView(height, width);
57
58          Container contents = getContentPane();
59
60          JPanel infoPane = new JPanel(new BorderLayout());
61              infoPane.add(stepLabel, BorderLayout.WEST);
62              infoPane.add(infoLabel, BorderLayout.CENTER);
63          contents.add(infoPane, BorderLayout.NORTH);
64          contents.add(fieldView, BorderLayout.CENTER);
65          JPanel downPane = new JPanel(new BorderLayout());
66          contents.add(downwardinfo, BorderLayout.SOUTH);
67          pack();
68          setVisible(true);
69      }
70
71      /**
72       * Define a color to be used for a given class of animal.
73       * @param animalClass The animal's Class object.
74       * @param color The color to be used for the given class.
75       */
76      public void setColor(Class animalClass, Color color)
77      {
78          colors.put(animalClass, color);
79      }
80
81      /**
82       * Display a short information label at the top of the window.
83       */
84      public void setInfoText(String text)
85      {
86          infoLabel.setText(text);
87      }
88
89      /**
90       * @return The color to be used for a given class of animal.
91       */
92      private Color getColor(Class animalClass)
93      {
94          Color col = colors.get(animalClass);
95          if(col == null)
96          {
97              // no color defined for this class
98              return UNKNOWN_COLOR;
99          }
100         else
101         {
102             return col;
103         }
104     }
105
106     /**
107      * Show the current status of the field.
108      * @param step Which iteration step it is.
```

```
     * @param field The field whose status is to be displayed.
     */
    public void showStatus(int step, Field field)
    {
        if(!isVisible())
        {
            setVisible(true);
        }

        stepLabel.setText(STEP_PREFIX + step);
        stats.reset();
        //String for weather condition
        String weather;
        if (Condition.getWeather() == 0)
            weather = "drought";
        else if (Condition.getWeather() == 1)
            weather = "normal";
        else
            weather = "wet";

        fieldView.preparePaint();

        for(int row = 0; row < field.getDepth(); row++)
        {
            for(int col = 0; col < field.getWidth(); col++)
            {
                Object animal = field.getObjectAt(row, col);
                if(animal != null)
                {
                    stats.incrementCount(animal.getClass());
                    fieldView.drawMark(col, row, getColor(animal.getClass()));
                }
                else
                {
                    fieldView.drawMark(col, row, EMPTY_COLOR);
                }
            }
        }
        stats.countFinished();
        //add time and weather information at the bottom of the window.
        downwardinfo.setText(POPULATION_PREFIX +
stats.getPopulationDetails(field)+ WEATHER_PREFIX + weather + " " + TIME_PREFIX +
Condition.getHours());
        fieldView.repaint();
    }

    /**
     * Determine whether the simulation should continue to run.
     * @return true If there is more than one species alive.
     */
    public boolean isViable(Field field)
    {
        return stats.isViable(field);
    }
```

```
161
162     /**
163      * Provide a graphical view of a rectangular field. This is
164      * a nested class (a class defined inside a class) which
165      * defines a custom component for the user interface. This
166      * component displays the field.
167      * This is rather advanced GUI stuff - you can ignore this
168      * for your project if you like.
169      */
170     private class FieldView extends JPanel
171     {
172         private final int GRID_VIEW_SCALING_FACTOR = 6;
173
174         private int gridWidth, gridHeight;
175         private int xScale, yScale;
176         Dimension size;
177         private Graphics g;
178         private Image fieldImage;
179
180         /**
181          * Create a new FieldView component.
182          */
183         public FieldView(int height, int width)
184         {
185             gridHeight = height;
186             gridWidth = width;
187             size = new Dimension(0, 0);
188         }
189
190         /**
191          * Tell the GUI manager how big we would like to be.
192          */
193         public Dimension getPreferredSize()
194         {
195             return new Dimension(gridWidth * GRID_VIEW_SCALING_FACTOR,
196                                  gridHeight * GRID_VIEW_SCALING_FACTOR);
197         }
198
199         /**
200          * Prepare for a new round of painting. Since the component
201          * may be resized, compute the scaling factor again.
202          */
203         public void preparePaint()
204         {
205             if(! size.equals(getSize()))
206             {  // if the size has changed...
207                 size = getSize();
208                 fieldImage = fieldView.createImage(size.width, size.height);
209                 g = fieldImage.getGraphics();
210
211                 xScale = size.width / gridWidth;
212                 if(xScale < 1)
213                 {
214                     xScale = GRID_VIEW_SCALING_FACTOR;
```

```
            }
            yScale = size.height / gridHeight;
            if(yScale < 1)
            {
                yScale = GRID_VIEW_SCALING_FACTOR;
            }
        }
    }

    /**
     * Paint on grid location on this field in a given color.
     */
    public void drawMark(int x, int y, Color color)
    {
        g.setColor(color);
        g.fillRect(x * xScale, y * yScale, xScale-1, yScale-1);
    }

    /**
     * The field view component needs to be redisplayed. Copy the
     * internal image to screen.
     */
    public void paintComponent(Graphics g)
    {
        if(fieldImage != null)
        {
            Dimension currentSize = getSize();
            if(size.equals(currentSize))
            {
                g.drawImage(fieldImage, 0, 0, null);
            }
            else
            {
                // Rescale the previous image.
                g.drawImage(fieldImage, 0, 0, currentSize.width,
currentSize.height, null);
            }
        }
    }
}

```

```java
import java.awt.Color;
import java.util.HashMap;

/**
 * This class collects and provides some statistical data on the state
 * of a field. It is flexible: it will create and maintain a counter
 * for any class of object that is found within the field.
 *
 * @author David J. Barnes and Michael Kölling
 * @version 2016.02.29
 */
public class FieldStats
{
    // Counters for each type of entity (fox, rabbit, etc.) in the simulation.
    private HashMap<Class, Counter> counters;
    // Whether the counters are currently up to date.
    private boolean countsValid;

    /**
     * Construct a FieldStats object.
     */
    public FieldStats()
    {
        // Set up a collection for counters for each type of animal that
        // we might find
        counters = new HashMap<>();
        countsValid = true;
    }

    /**
     * Get details of what is in the field.
     * @return A string describing what is in the field.
     */
    public String getPopulationDetails(Field field)
    {
        StringBuffer buffer = new StringBuffer();
        if(!countsValid)
        {
            generateCounts(field);
        }
        for(Class key : counters.keySet())
        {
            Counter info = counters.get(key);
            buffer.append(info.getName());
            buffer.append(": ");
            buffer.append(info.getCount());
            buffer.append(' ');
        }
        return buffer.toString();
    }

    /**
     * Invalidate the current set of statistics; reset all
     * counts to zero.
```

```
 */
public void reset()
{
    countsValid = false;
    for(Class key : counters.keySet())
    {
        Counter count = counters.get(key);
        count.reset();
    }
}

/**
 * Increment the count for one class of animal.
 * @param animalClass The class of animal to increment.
 */
public void incrementCount(Class animalClass)
{
    Counter count = counters.get(animalClass);
    if(count == null)
    {
        // We do not have a counter for this species yet.
        // Create one.
        count = new Counter(animalClass.getName());
        counters.put(animalClass, count);
    }
    count.increment();
}

/**
 * Indicate that an animal count has been completed.
 */
public void countFinished()
{
    countsValid = true;
}

/**
 * Determine whether the simulation is still viable.
 * I.e., should it continue to run.
 * @return true If there is more than one species alive.
 */
public boolean isViable(Field field)
{
    // How many counts are non-zero.
    int nonZero = 0;
    if(!countsValid)
    {
        generateCounts(field);
    }
    for(Class key : counters.keySet())
    {
        Counter info = counters.get(key);
        if(info.getCount() > 0)
        {
```

```
                nonZero++;
            }
        }
        return nonZero > 1;
    }

    /**
     * Generate counts of the number of creatures.
     * These are not kept up to date as creatures.
     * are placed in the field, but only when a request
     * is made for the information.
     * @param field The field to generate the stats for.
     */
    private void generateCounts(Field field)
    {
        reset();
        for(int row = 0; row < field.getDepth(); row++)
        {
            for(int col = 0; col < field.getWidth(); col++)
            {
                Object animal = field.getObjectAt(row, col);
                if(animal != null)
                {
                    incrementCount(animal.getClass());
                }
            }
        }
        countsValid = true;
    }
}
```

```java
import java.util.Collections;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Random;
import javafx.util.Pair;
import java.util.HashMap;

/**
 * Represent a rectangular grid of field positions.
 * Each position is able to store a single animal.
 *
 * @author David J. Barnes, Michael Kölling, Houkang Zhang and Yeshuai Cui
 * @version 2020/02/20
 */
public class Field
{
    // A random number generator for providing random locations.
    private static final Random rand = Randomizer.getRandom();
    // The depth and width of the field.
    private int depth, width;
    // Storage for the animals.
    private Object[][] field;

    /**
     * Represent a field of the given dimensions.
     * @param depth The depth of the field.
     * @param width The width of the field.
     */
    public Field(int depth, int width)
    {
        this.depth = depth;
        this.width = width;
        field = new Object[depth][width];
    }

    /**
     * Empty the field.
     */
    public void clear()
    {
        for(int row = 0; row < depth; row++)
        {
            for(int col = 0; col < width; col++)
            {
                field[row][col] = null;
            }
        }
    }

    /**
     * Clear the given location.
     * @param location The location to clear.
     */
```

```
55  public void clear(Location location)
56  {
57      field[location.getRow()][location.getCol()] = null;
58  }
59
60  /**
61   * Place an animal at the given location.
62   * If there is already an animal at the location it will
63   * be lost.
64   * @param animal The animal to be placed.
65   * @param row Row coordinate of the location.
66   * @param col Column coordinate of the location.
67   */
68  public void place(Object animal, int row, int col)
69  {
70      place(animal, new Location(row, col));
71  }
72
73  /**
74   * Place an animal at the given location.
75   * If there is already an animal at the location it will
76   * be lost.
77   * @param animal The animal to be placed.
78   * @param location Where to place the animal.
79   */
80  public void place(Object animal, Location location)
81  {
82      field[location.getRow()][location.getCol()] = animal;
83  }
84
85  /**
86   * Return the animal at the given location, if any.
87   * @param location Where in the field.
88   * @return The animal at the given location, or null if there is none.
89   */
90  public Object getObjectAt(Location location)
91  {
92      return getObjectAt(location.getRow(), location.getCol());
93  }
94
95  /**
96   * Return the animal at the given location, if any.
97   * @param row The desired row.
98   * @param col The desired column.
99   * @return The animal at the given location, or null if there is none.
100  */
101 public Object getObjectAt(int row, int col)
102 {
103     return field[row][col];
104 }
105
106 /**
107  * Generate a random location that is adjacent to the
108  * given location, or is the same location.
```

```
109     * The returned location will be within the valid bounds
110     * of the field.
111     * @param location The location from which to generate an adjacency.
112     * @return A valid location within the grid area.
113     */
114    public Location randomAdjacentLocation(Location location)
115    {
116        List<Location> adjacent = adjacentLocations(location);
117        return adjacent.get(0);
118    }
119
120    /**
121     * Get a shuffled list of the free adjacent locations.
122     * @param location Get locations adjacent to this.
123     * @return A list of free adjacent locations.
124     */
125    public List<Location> getFreeAdjacentLocations(Location location)
126    {
127        List<Location> free = new LinkedList<>();
128        List<Location> adjacent = adjacentLocations(location);
129        for(Location next : adjacent)
130        {
131            if(getObjectAt(next) == null)
132            {
133                free.add(next);
134            }
135        }
136        return free;
137    }
138
139    /**
140     * Try to find a free location that is adjacent to the
141     * given location. If there is none, return null.
142     * The returned location will be within the valid bounds
143     * of the field.
144     * @param location The location from which to generate an adjacency.
145     * @return A valid location within the grid area.
146     */
147    public Location freeAdjacentLocation(Location location)
148    {
149        // The available free ones.
150        List<Location> free = getFreeAdjacentLocations(location);
151        if(free.size() > 0)
152        {
153            return free.get(0);
154        }
155        else
156        {
157            return null;
158        }
159    }
160
161    /**
162     * Return a shuffled list of locations adjacent to the given one.
```

```
     * The list will not include the location itself.
     * All locations will lie within the grid.
     * @param location The location from which to generate adjacencies.
     * @return A list of locations adjacent to that given.
     */
    public List<Location> adjacentLocations(Location location)
    {
        assert location != null : "Null location passed to adjacentLocations";
        // The list of locations to be returned.
        List<Location> locations = new LinkedList<>();
        if(location != null)
        {
            int row = location.getRow();
            int col = location.getCol();
            for(int roffset = -1; roffset <= 1; roffset++)
            {
                int nextRow = row + roffset;
                if(nextRow >= 0 && nextRow < depth)
                {
                    for(int coffset = -1; coffset <= 1; coffset++)
                    {
                        int nextCol = col + coffset;
                        // Exclude invalid locations and the original location.
                        if(nextCol >= 0 && nextCol < width && (roffset != 0 ||
coffset != 0))
                        {
                            locations.add(new Location(nextRow, nextCol));
                        }
                    }
                }
            }

            // Shuffle the list. Several other methods rely on the list
            // being in a random order.
            Collections.shuffle(locations, rand);
        }
        return locations;
    }

    /**
     * Return the depth of the field.
     * @return The depth of the field.
     */
    public int getDepth()
    {
        return depth;
    }

    /**
     * Return the width of the field.
     * @return The width of the field.
     */
    public int getWidth()
    {
```

```
216             return width;
217         }
218     }
219
```

```java
import java.awt.Color;

/**
 * Provide a counter for a participant in the simulation.
 * This includes an identifying string and a count of how
 * many participants of this type currently exist within
 * the simulation.
 *
 * @author David J. Barnes, Michael Kölling, Houkang Zhang and Yeshuai Cui
 * @version 2020/02/20
 */
public class Counter
{
    // A name for this type of simulation participant
    private String name;
    // How many of this type exist in the simulation.
    private int count;

    /**
     * Provide a name for one of the simulation types.
     * @param name  A name, e.g. "Grass".
     */
    public Counter(String name)
    {
        this.name = name;
        count = 0;
    }

    /**
     * @return The short description of this type.
     */
    public String getName()
    {
        return name;
    }

    /**
     * @return The current count for this type.
     */
    public int getCount()
    {
        return count;
    }

    /**
     * Increment the current count by one.
     */
    public void increment()
    {
        count++;
    }

    /**
     * Reset the current count to zero.
```

```
55      */
56      public void reset()
57      {
58          count = 0;
59      }
60  }
61
```

```java
import java.util.*;
import java.util.Random;
import javafx.util.Pair;
/**
 * this class is to represent common characteristics and acts for animals
 */
public abstract class Animal extends Creature
{
    //The gender of the animal
    protected boolean isMale;
    //Whether the animal is infected by disease
    protected boolean ill = false;
    //Random generator
    private static final Random rand = Randomizer.getRandom();
    //The probability that this animal is the source of disease.
    protected static final double ILL_PROBABILITY = 0.005;
    //The probability that infected animal can cure themselves is one step.
    protected static final double CURE_PROBABILITY = 0.01;
    //the steps take of an infected animal die.
    protected static final int KILLED_BY_DISEASE = 7;
    //the gender probability is even as male or female
    private static final double MALE_PROBABILITY = 0.5;
    //all the animals are created with zero ill level
    protected int illLevel = 0;
    //current foodlevel of animals, die if it reaches 0
    protected int foodLevel;

    /**
     * @param Field and location of this animal
     */
    public Animal(Field field , Location location )
    {
        super(field , location);
        double ran = rand.nextDouble();
        //gender of new born animal
        if(ran <= MALE_PROBABILITY)
        {
            isMale = true;
        }
        else
        {
            isMale = false;
        }
    }

    /**
     * @return boolean value true for male and false for female
     */
    protected boolean getGender()
    {
        return isMale;
    }

    /**
```

```
     * this animal is infected by the disease
     */
    protected void setIll()
    {
        ill = true;
    }

    /**
     * the animal is cured.
     */
    protected void cure()
    {
        ill = false;
        illLevel = 0;
    }

    /**
     * @return boolean value whether animal is infected by the disease
     */
    protected boolean isIll()
    {
        return ill;
    }

    /**
     * Check if illLevel reach the limit and kill the animal,
     * and there's a possibility an animal is cured by itself.
     */
    protected void illness()
    {
        if(isAlive())
        {
            if(isIll())
            {
                illLevel++;
                if(illLevel > KILLED_BY_DISEASE)
                {
                setDead();
                }
                else
                {
                findInfect();
                if(rand.nextDouble() < CURE_PROBABILITY)
                {
                    cure();
                }
            }
        }
            else
            {
                if(rand.nextDouble() < ILL_PROBABILITY)
                {
                    setIll();
                }
```

```
109             }
110         }
111         else
112             return;
113     }
114
115     /**
116      * @return a iterator of locations around where this animal is located
117      */
118     protected Iterator<Location> adjacentIterator()
119     {
120         List<Location> adjacent = field.adjacentLocations(getLocation());
121         Iterator<Location> it = adjacent.iterator();
122         return it;
123     }
124
125     /**
126      * Make this Dragonfly more hungry. This could result in the Dragonfly's
    death.
127      */
128     protected void incrementHunger()
129     {
130         foodLevel--;
131         if(foodLevel <= 0)
132         {
133             setDead();
134         }
135     }
136     public abstract void act(List<Creature> animal);
137
138     public abstract void findInfect();
139 }
140
```

```java
import java.util.*;
import javafx.util.Pair;

/**
 * A simple model of a Sparrow.
 * Sparrow age, move, eat butterfly, grasshopper and dragonfly, it may infected by
 * disease and die because of disease and starvation. Condition will influence
 * the act of sparrow
 * @author David J. Barnes,  Michael Kölling, Houkang Zhang and Yeshuai Cui
 * @version 2020/02/20
 */
public class Sparrow extends Animal
{
    // Characteristics shared by all Sparrows (class variables).
    // The age at which a sparrow can start to breed.
    private static final int BREEDING_AGE =10;
    // The age to which a sparrow can live.
    private static final int MAX_AGE = 300;
    // The likelihood of a sparrow breeding.
    private static final double BREEDING_PROBABILITY = 0.7;
    // The maximum number of births.
    private static final int MAX_LITTER_SIZE = 3;
    // The food value of a single dragonfly. In effect, this is the
    // number of steps a sparrow can go before it has to eat again.
    private static final int DRAGONFLY_FOOD_VALUE = 40;
    // The food value of a single butterfly.
    private static final int BUTTERFLY_FOOD_VALUE = 40;
    // The food value of a single grasshoper.
    private static final int GRASSHOPPER_FOOD_VALUE = 40;
    // A shared random number generator to control breeding.
    private static final Random rand = Randomizer.getRandom();
    //The possibility that this step will do nothing
    private static final double REST_PROBABILITY = 0.3;
    // Individual characteristics (instance fields).
    /**
     * Create a sparrow. A sparrow can be created as a new born (age zero
     * and not hungry) or with a random age and food level.
     *
     * @param randomAge If true, the Dragonfly will have random age and hunger level.
     * @param field The field currently occupied.
     * @param location The location within the field.
     */
    public Sparrow(boolean randomAge, Field field, Location location)
    {
        super(field, location);
        if(randomAge) {
            age = rand.nextInt(MAX_AGE);
            //food level of new created sparrow
            foodLevel = rand.nextInt(DRAGONFLY_FOOD_VALUE);
        }
        else {
            age = 0;
```

```
53              //food level of new born sparrow
54              foodLevel = (int)(DRAGONFLY_FOOD_VALUE * 0.7);
55          }
56      }
57
58      /**
59       * This is what the sparrow does most of the time: it hunts for
60       * butterfly, dragonfly and grasshopper. In the process, it might breed, die
    because
61       * of starvation, age or disease.
62       * @param field The field currently occupied.
63       * @param newDragonflies A list to return newly born sparrows.
64       */
65      public void act(List<Creature> newSparrow)
66      {
67          incrementAge();
68          if(rand.nextDouble() < REST_PROBABILITY)
69          {
70              return;
71          }
72          else
73          {
74              incrementHunger();
75              illness();
76              if(isAlive())
77              {
78                  giveBirth(newSparrow);
79                  // Move towards a source of food if found.
80                  Location newLocation = findFood();
81                  if(newLocation == null)
82                  {
83                      // No food found - try to move to a free location.
84                      newLocation = getField().freeAdjacentLocation(getLocation());
85                  }
86                  // See if it was possible to move.
87                  if(newLocation != null)
88                  {
89                      setLocation(newLocation);
90                  }
91                  else
92                  {
93                      // Overcrowding.
94                      setDead();
95                  }
96              }
97          }
98      }
99
100     /**
101      * Increase the age. This could result in the sparrow's death.
102      */
103     private void incrementAge()
104     {
105         age++;
```

```
106         if(age > MAX_AGE) {
107             setDead();
108         }
109     }
110
111     /**
112      * Look for preys adjacent to the current location.
113      * Only the first live prey is eaten.
114      * @return Where food was found, or null if it wasn't.
115      */
116     private Location findFood()
117     {
118         Iterator<Location> it = adjacentIterator();
119         while(it.hasNext())
120         {
121             Location where = it.next();
122             Object what = field.getObjectAt(where);
123             if(what instanceof Grasshopper)
124             {
125                 Grasshopper grasshopper = (Grasshopper) what ;
126                 if(grasshopper.isAlive())
127                 {
128                     grasshopper.setDead();
129                     foodLevel = GRASSHOPPER_FOOD_VALUE;
130                     return where;
131                 }
132             }
133             else if(what instanceof Butterfly)
134             {
135                 Butterfly butterfly = (Butterfly) what;
136                 if(butterfly.isAlive())
137                 {
138                     butterfly.setDead();
139                     foodLevel = BUTTERFLY_FOOD_VALUE;
140                     return where;
141                 }
142             }
143             else if(what instanceof Dragonfly)
144             {
145                 Dragonfly dragonfly = (Dragonfly) what;
146                 if(dragonfly.isAlive())
147                 {
148                     dragonfly.setDead();
149                     foodLevel = DRAGONFLY_FOOD_VALUE;
150                     return where;
151                 }
152             }
153         }
154         return null;
155     }
156
157     /**
158      * If sparrow is ill it may spread disease to adjacent sparrows
159      */
```

```
160     public void findInfect()
161     {
162         Iterator<Location> it = adjacentIterator();
163         while(it.hasNext()) {
164             Location where = it.next();
165             Object animal = field.getObjectAt(where);
166             if( rand.nextDouble() <= ILL_PROBABILITY &&isIll() && animal
    instanceof Sparrow)
167             {
168                 Sparrow sparrow = (Sparrow) animal;
169                 if(sparrow.isAlive())
170                 {
171                     sparrow.setIll();
172                 }
173             }
174         }
175     }
176
177     /**
178      * Check whether or not this sprrow is to give birth at this step.
179      * New births will be made into free adjacent locations.
180      * @param newSparrow A list to return newly born Sparrow.
181      */
182     private void giveBirth(List<Creature> newSparrow)
183     {
184         // New Dragonflyes are born into adjacent locations.
185         // Get a list of adjacent free locations.
186         // This method is called by female fragonfly
187         if(!getGender())
188         {
189             Field field = getField();
190             List<Location> adjacent = field.adjacentLocations(getLocation());
191             Iterator<Location> it = adjacent.iterator();
192             while(it.hasNext())
193             {
194                 Location where = it.next();
195                 Object what = field.getObjectAt(where);
196                 if(what instanceof Sparrow)
197                 {
198                     Sparrow sparrow = (Sparrow) what ;
199                     //only reproduce when two sparrow are of opposite gender
200                     if(sparrow.getGender())
201                     {
202                         List<Location> free =
    field.getFreeAdjacentLocations(getLocation());
203                         int births = breed();
204                         for(int b = 0; b < births && free.size() > 0; b++) {
205                             Location loc = free.remove(0);
206                             Sparrow young = new Sparrow(false, field, loc);
207                             newSparrow.add(young);
208                             break;
209                           }
210                     }
211                 }
```

```
        }
    }
}

/**
 * Generate a number representing the number of births,
 * if it can breed.
 * @return The number of births (may be zero).
 */
private int breed()
{
    int births = 0;
    if(canBreed() && rand.nextDouble() <= BREEDING_PROBABILITY && foodLevel > 9)
    {
        births = rand.nextInt(MAX_LITTER_SIZE) + 1;//inclusive
    }
    if(births > 0)
    {
        foodLevel -= 1 ;//sparrow will lose foodlevel when give birth
        //to its offsprings.
    }

    return births;
}

/**
 * @return boolean
 * A Dragonfly can breed if it has reached the breeding age.
 */
private boolean canBreed()
{
    return age >= BREEDING_AGE;
}
}
```

```java
import java.util.*;
import javafx.util.Pair;

/**
 * A simple model of a Eagle.
 * Eagles age, move, eat sparrow, it may infected by
 * disease and die because of disease and starvation. Condition will influence
 * the act of eagle
 * *@author David J. Barnes,  Michael Kölling, Houkang Zhang and Yeshuai Cui
 * @version 2020/02/20
 */
public class Eagle extends Animal
{
    // Characteristics shared by all eagles (class variables).
    // The age at which a eagle can start to breed.
    private static final int BREEDING_AGE = 30;
    // The age to which a eagle can live.
    private static final int MAX_AGE = 1000;
    // The likelihood of a eagle breeding.
    private static final double BREEDING_PROBABILITY = 0.43;
    // The maximum number of births.
    private static final int MAX_LITTER_SIZE = 1;
    // The food value of a single eagle. In effect, this is the
    // number of steps a eagle can go before it has to eat again.
    private static final int SPARROW_FOOD_VALUE = 80;
    // A shared random number generator to control breeding.
    private static final Random rand = Randomizer.getRandom();
    // The probability that eagle will move 2 times at nignt
    private static double NIGHT_MOVEMENT_PROBABILITY = 0.5;

    /**
     * Create a eagle. A eagle can be created as a new born (age zero
     * and not hungry) or with a random age and food level.
     *
     * @param randomAge If true, the eagle will have random age and hunger level.
     * @param field The field currently occupied.
     * @param location The location within the field.
     */
    public Eagle(boolean randomAge, Field field, Location location)
    {
        super(field, location);
        if(randomAge)
        {
            age = rand.nextInt(MAX_AGE);
            foodLevel = rand.nextInt(SPARROW_FOOD_VALUE);
        }
        else
        {
            age = 0;
            foodLevel = (int)(SPARROW_FOOD_VALUE/0.6);
        }
    }

    /**
```

```
     * This is what the eagle does most of the time: it hunts for
     * sparrow. In the process, it might breed, die because
     * of starvation, age or disease.
     * @param field The field currently occupied.
     * @param newEagle A list to return newly born eagle.
     */
    public void act(List<Creature> newEagles)
    {
        incrementAge();
        incrementHunger();
        illness();
        int motion = 1;
        if(!Condition.getIfDay() && Condition.getWeather() == 0 &&
rand.nextDouble() <= NIGHT_MOVEMENT_PROBABILITY)
        {
            motion = 2;
        }
        for(int i = 0; i <= motion; i++)
        {
            if(isAlive())
            {
                giveBirth(newEagles);
                // Move towards a source of food if found.
                Location newLocation = findFood();
                if(newLocation == null)
                {
                    // No food found - try to move to a free location.
                    newLocation = getField().freeAdjacentLocation(getLocation());
                }
                // See if it was possible to move.
                if(newLocation != null)
                {
                    setLocation(newLocation);
                }
                else
                {
                    // Overcrowding.
                    setDead();
                }
            }
        }
    }

    /**
     * Increase the age. This could result in the eagle's death.
     */
    private void incrementAge()
    {
        age++;
        if(age > MAX_AGE) {
            setDead();
        }
    }

```

```
108     /**
109      * Look for sparrows adjacent to the current location.
110      * Only the first live sparrow is eaten.
111      * @return Where food was found, or null if it wasn't.
112      */
113     private Location findFood()
114     {
115         Iterator<Location> it = adjacentIterator();
116         while(it.hasNext())
117         {
118             Location where = it.next();
119             Object animal = field.getObjectAt(where);
120             if(animal instanceof Sparrow)
121             {
122                 Sparrow sparrow = (Sparrow) animal;
123                 if(sparrow.isAlive())
124                 {
125                     sparrow.setDead();
126                     foodLevel = SPARROW_FOOD_VALUE;
127                     return where;
128                 }
129             }
130         }
131         return null;
132     }
133
134     /**
135      * If eagle is ill it may spread disease to adjacent eagle
136      */
137     public void findInfect()
138     {
139         Iterator<Location> it = adjacentIterator();
140         while(it.hasNext())
141         {
142             Location where = it.next();
143             Object animal = field.getObjectAt(where);
144             if( rand.nextDouble() <= ILL_PROBABILITY &&isIll() && animal
    instanceof Eagle)
145             {
146                 Eagle eagle = (Eagle) animal;
147                 if(eagle.isAlive())
148                 {
149                     eagle.setIll();
150                 }
151             }
152         }
153     }
154
155     /**
156      * Check whether or not this eagle is to give birth at this step.
157      * New births will be made into free adjacent locations.
158      * @param newEagle A list to return newly born eagle.
159      */
160     private void giveBirth(List<Creature> newEagle)
```

```
161     {
162         // New eagle are born into adjacent locations.
163         // Get a list of adjacent free locations.
164         // This method is called by female eagle
165         if(!getGender())
166         {
167             Field field = getField();
168             List<Location> adjacent = field.adjacentLocations(getLocation());
169             Iterator<Location> it = adjacent.iterator();
170             while(it.hasNext())
171             {
172                 Location where = it.next();
173                 Object what = field.getObjectAt(where);
174                 if(what instanceof Eagle)
175                 {
176                     Eagle eagle = (Eagle) what ;
177                     //only reproduce when two eagle are of opposite gender
178                     if(eagle.getGender())
179                     {
180                         List<Location> free =
    field.getFreeAdjacentLocations(getLocation());
181                         int births = breed();
182                         for(int b = 0; b < births && free.size() > 0; b++)
183                         {
184                             Location loc = free.remove(0);
185                             Eagle young = new Eagle(false, field, loc);
186                             newEagle.add(young);
187                             break;
188                         }
189                     }
190                 }
191             }
192         }
193     }
194
195     /**
196      * Generate a number representing the number of births,
197      * if it can breed.
198      * @return The number of births (may be zero).
199      */
200     private int breed()
201     {
202         int births = 0;
203         if(canBreed() && rand.nextDouble() <= BREEDING_PROBABILITY && foodLevel >
    10)
204         {
205             births = rand.nextInt(MAX_LITTER_SIZE) + 1;
206         }
207         if(births > 0)
208         {
209             foodLevel -= 1 ;
210         }
211         return births;
212     }
```

```
213
214     /**
215      * A eagle can breed if it has reached the breeding age.
216      */
217     private boolean canBreed()
218     {
219         return age >= BREEDING_AGE;
220     }
221 }
```

```java
import java.util.Random;
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;
import java.awt.Color;

/**
 * A simple predator-prey simulator, based on a rectangular field
 * containing different animals and grass as a food source.
 *
 * @author David J. Barnes,  Michael Kölling, Houkang Zhang and Yeshuai Cui
 * @version 2020/02/20
 */
public class Simulator
{
    // Constants representing configuration information for the simulation.
    // The default width for the grid.
    private static final int DEFAULT_WIDTH = 250;
    // The default depth of the grid.
    private static final int DEFAULT_DEPTH = 250;
    // The probability that a dragonfly will be created in any given grid position.
    private static final double DRAGONFLY_CREATION_PROBABILITY = 0.04;
    // The probability that a butterfly will be created in any given grid position.
    private static final double BUTTERFLY_CREATION_PROBABILITY = 0.071;
    // The probability that a grasshopper will be created in any given grid position.
    private static final double GRASSHOPPER_CREATION_PROBABILITY = 0.05;
    // The probability that a sparrow will be created in any given grid position.
    private static final double SPARROW_CREATION_PROBABILITY = 0.04;
    // The probability that a grass will be created in any given grid position.
    private static final double GRASS_CREATION_PROBABILITY = 0.11;
    // The probability that a eagle will be created in any given grid position.
    private static final double EAGLE_CREATION_PROBABILITY = 0.012;
    // List of Creatures in the field.
    private List<Creature> creatures;
    // The current state of the field.
    private Field field;
    // The current step of the simulation.
    private int step;
    // A graphical view of the simulation.
    private SimulatorView view;

    /**
     * Construct a simulation field with default size.
     */
    public Simulator()
    {
        this(DEFAULT_DEPTH, DEFAULT_WIDTH);
    }

    /**
     * Create a simulation field with the given size.
```

```
52      * @param depth Depth of the field. Must be greater than zero.
53      * @param width Width of the field. Must be greater than zero.
54      */
55     public Simulator(int depth, int width)
56     {
57         if(width <= 0 || depth <= 0) {
58             System.out.println("The dimensions must be greater than zero.");
59             System.out.println("Using default values.");
60             depth = DEFAULT_DEPTH;
61             width = DEFAULT_WIDTH;
62         }
63 
64         creatures = new ArrayList<>();
65         field = new Field(depth, width);
66 
67         // Create a view of the state of each location in the field.
68         view = new SimulatorView(depth, width);
69         view.setColor(Butterfly.class, Color.RED);
70         view.setColor(Dragonfly.class, Color.BLUE);
71         view.setColor(Grasshopper.class , Color.PINK);
72         view.setColor(Grass.class , Color.green);
73         view.setColor(Sparrow.class , Color.orange);
74         view.setColor(Eagle.class , Color.black);
75         // Setup a valid starting point.
76         reset();
77     }
78 
79     /**
80      * Run the simulation from its current state for a reasonably long period,
81      * (4000 steps).
82      */
83     public void runLongSimulation()
84     {
85         simulate(4000);
86     }
87 
88     /**
89      * Run the simulation from its current state for the given number of steps.
90      * Stop before the given number of steps if it ceases to be viable.
91      * @param numSteps The number of steps to run for.
92      */
93     public void simulate(int numSteps)
94     {
95         for(int step = 1; step <= numSteps && view.isViable(field); step++)
96         {
97             simulateOneStep();
98             //delay(60);// uncomment this to run more slowly
99         }
100    }
101 
102    /**
103     * Run the simulation from its current state for a single step.
104     * Iterate over the whole field updating the state of each creature
105     */
```

```
106     public void simulateOneStep()
107     {
108         step++;
109         Condition.update();
110         // Provide space for newborn Creatures.
111         List<Creature> newCreatures = new ArrayList<>();
112         // Let all rabbits act.
113         for(Iterator<Creature> it = creatures.iterator(); it.hasNext(); )
114         {
115             Creature creature = it.next();
116             creature.act(newCreatures);
117             if(! creature.isAlive())
118             {
119                 it.remove();
120             }
121         }
122
123         // Add the newly born foxes and rabbits to the main lists.
124         creatures.addAll(newCreatures);
125
126         view.showStatus(step, field);
127     }
128
129     /**
130      * Reset the simulation to a starting position.
131      */
132     public void reset()
133     {
134         step = 0;
135         creatures.clear();
136         populate();
137
138         // Show the starting state in the view.
139         view.showStatus(step, field);
140     }
141
142     /**
143      * Randomly populate the field with creatures.
144      */
145     private void populate()
146     {
147         Random rand = Randomizer.getRandom();
148         field.clear();
149         for(int row = 0; row < field.getDepth(); row++)
150         {
151             for(int col = 0; col < field.getWidth(); col++)
152             {
153                 if(rand.nextDouble() <= DRAGONFLY_CREATION_PROBABILITY)
154                 {
155                     Location location = new Location(row, col);
156                     Dragonfly drangonfly = new Dragonfly(true, field, location);
157                     creatures.add(drangonfly);
158                 }
159                 else if(rand.nextDouble() <= BUTTERFLY_CREATION_PROBABILITY)
```

```
                {
                    Location location = new Location(row, col);
                    Butterfly butterfly = new Butterfly(true, field, location);
                    creatures.add(butterfly);
                }
                else if(rand.nextDouble() <= GRASSHOPPER_CREATION_PROBABILITY)
                {
                    Location location = new Location(row , col);
                    Grasshopper grasshopper = new Grasshopper(true , field ,
location);
                    creatures.add(grasshopper);
                }
                else if(rand.nextDouble() <= GRASS_CREATION_PROBABILITY)
                {
                    Location location = new Location(row , col);
                    Grass grass = new Grass(true , field , location);
                    creatures.add(grass);
                }
                else if(rand.nextDouble() <= SPARROW_CREATION_PROBABILITY)
                {
                    Location location = new Location(row , col);
                    Sparrow sparrow = new Sparrow(true , field , location);
                    creatures.add(sparrow);
                }
                else if(rand.nextDouble() <= EAGLE_CREATION_PROBABILITY)
                {
                    Location location = new Location(row ,col);
                    Eagle eagle = new Eagle(true , field , location);
                    creatures.add(eagle);
                }
                // else leave the location empty.
            }
        }
    }

    /**
     * Pause for a given time.
     * @param millisec  The time to pause for, in milliseconds
     */
    private void delay(int millisec)
    {
        try
        {
            Thread.sleep(millisec);
        }
        catch (InterruptedException ie)
        {
            // wake up
        }
    }
}
```

```java
/**
 * Represent a location in a rectangular grid.
 *
 * @author David J. Barnes and Michael Kölling
 * @version 2016.02.29
 */
public class Location
{
    // Row and column positions.
    private int row;
    private int col;

    /**
     * Represent a row and column.
     * @param row The row.
     * @param col The column.
     */
    public Location(int row, int col)
    {
        this.row = row;
        this.col = col;
    }

    /**
     * Implement content equality.
     */
    public boolean equals(Object obj)
    {
        if(obj instanceof Location)
        {
            Location other = (Location) obj;
            return row == other.getRow() && col == other.getCol();
        }
        else
        {
            return false;
        }
    }

    /**
     * Return a string of the form row,column
     * @return A string representation of the location.
     */
    public String toString()
    {
        return row + "," + col;
    }

    /**
     * Use the top 16 bits for the row value and the bottom for
     * the column. Except for very big grids, this should give a
     * unique hash code for each (row, col) pair.
     * @return A hashcode for the location.
     */
```

```
    public int hashCode()
    {
        return (row << 16) + col;
    }

    /**
     * @return The row.
     */
    public int getRow()
    {
        return row;
    }

    /**
     * @return The column.
     */
    public int getCol()
    {
        return col;
    }
}
```

```
Project: PPA assignment 3
Authors: Michael Kölling, David J. Barnes, Houkang Zhang and Yeshuai Cui

A predator-prey simulation involving grass and animals in
an enclosed rectangular field.

How to start:
    Create a Simulator object.
    Then call one of:
        + simulateOneStep - for a single step.
        + simulate - and supply a number (say 10) for that many steps.
        + runLongSimulation - for a simulation of 500 steps.

```