Additional Abstraction Techniques

• The material in this chapter centers around two versions of the Foxes and Rabbits project
  – In version 1 the Fox class and the Rabbit class are separate
  – The fox and rabbit share some behavior (birth, death, etc.) that can be abstracted into a parent class called Animal; this is implemented as version 2

• Other important topics
  – Processing information stored in a two dimensional array
  – Designing for maximum adaptability; in the Lab you will implement “The Game of Life” using a similar design to Foxes and Rabbits
Foxes and Rabbits

- Here is a picture of running the simulation
  - The blue (dark) squares are foxes
  - The orange (gray) squares are rabbits
  - The white squares are empty

- This simulation mimics predator-prey behavior found in nature
  - If there are too many predators there is not enough prey and the predators start dying from starvation
  - If there are not enough predators the prey start multiplying too rapidly and die due to lack of enough food in the environment
  - When in balance both populations cycle but remain persistent over time
Class Structure
Description of Classes

• The *Field* is a two dimensional grid where each cell can hold exactly one type of animal or be empty
• The *Location* is a position in the field as determined by row and column numbers
• There are only two animals, *Fox* and *Rabbit*
• The *Simulator* is the main program that creates the initial state and controls the step-to-step transition
• The *SimulatorView* is a graphical representation of the field and its contents
• The *FieldStats* keeps track of various values such as the number of foxes and rabbits present in the field
• The *Counter* stores the various counts associated with each object
• Our initial discussion concentrates on the *Rabbit, Fox* and *Simulator* classes
Some Details about Rabbits

• Some static variables shared by all rabbits: maximum age, breeding age, breeding probability, maximum litter size

• Instance variables include age, location, and a boolean to indicate if the rabbit is alive

• There is no attempt to model male and female behavior, all rabbits can breed once beyond the breeding age
import java.util.List;
import java.util.Random;

/**
 * A simple model of a rabbit.
 * Rabbits age, move, breed, and die.
 *
 * @author David J. Barnes and Michael Kolling
 * @version 2002-04-11
 */

public class Rabbit {

    // The age at which a rabbit can start to breed.
    private static final int BREEDING_AGE = 5;
    // The age to which a rabbit can live.
    private static final int MAX_AGE = 50;
    // The likelihood of a rabbit breeding.
    private static final double BREEDING_PROBABILITY = 0.15;
    // The maximum number of births.
    private static final int MAX_LITTER_SIZE = 5;
    // A shared random number generator to control breeding.
    private static final Random rand = new Random();
    // Individual characteristics (instance fields).
    // The rabbit's age.
    private int age;
    // Whether the rabbit is alive or not.
    private boolean alive;
    // The rabbit's position
    private Location location;
public Rabbit(boolean randomAge) {
    age = 0; alive = true;
    if(randomAge) { age = rand.nextInt(MAX_AGE); }
}

public void run(Field updatedField, List newRabbits) {
    incrementAge();
    if(alive) {
        int births = breed();
        for(int b = 0; b < births; b++) {
            Rabbit newRabbit = new Rabbit(false); newRabbits.add(newRabbit);
            Location loc = updatedField.randomAdjacentLocation(location);
            newRabbit.setLocation(loc);
            updatedField.place(newRabbit, loc);
        }
        Location newLocation = updatedField.freeAdjacentLocation(location);
        // Only transfer to the updated field if there was a free location
        if(newLocation != null) {
            setLocation(newLocation); updatedField.place(this, newLocation);
        } else {
            // can neither move nor stay - overcrowding - all locations taken
            alive = false;
        }
    }
}
private void incrementAge() {
    age++;
    if (age > MAX_AGE) {
        alive = false;
    }
}

private int breed() {
    int births = 0;
    if (canBreed() && rand.nextDouble() <= BREEDING_PROBABILITY) {
        births = rand.nextInt(MAX_LITTER_SIZE) + 1;
    }
    return births;
}

private boolean canBreed() {
    return age >= BREEDING_AGE;
}

public boolean isAlive() {
    return alive;
}
/**
 * Tell the rabbit that it's dead now :(
 * */
public void setEaten()
{
    alive = false;
}

/**
 * Set the animal's location.
 * @param row The vertical coordinate of the location.
 * @param col The horizontal coordinate of the location.
 */
public void setLocation(int row, int col)
{
    this.location = new Location(row, col);
}

/**
 * Set the rabbit's location.
 * @param location The rabbit's location.
 */
public void setLocation(Location location)
{
    this.location = location;
}
Some Details about Foxes

- Some static variables shared by all foxes: maximum age, breeding age, breeding probability, maximum litter size, and the food value for eating rabbits (you have to eat to stay alive!)
- Instance variables include age, location, an “alive” boolean value, and a food level
- There is no attempt to model male and female behavior, all foxes can breed once beyond the breeding age
- The foxes try to feed on the rabbits to stay alive
public class Fox
{
    // The age at which a fox can start to breed.
    private static final int BREEDING_AGE = 10;
    // The age to which a fox can live.
    private static final int MAX_AGE = 150;
    // The likelihood of a fox breeding.
    private static final double BREEDING_PROBABILITY = 0.09;
    // The maximum number of births.
    private static final int MAX_LITTER_SIZE = 3;
    // The food value of a single rabbit. In effect, this is the
    // number of steps a fox can go before it has to eat again.
    private static final int RABBIT_FOOD_VALUE = 4;
    // A shared random number generator to control breeding.
    private static final Random rand = new Random();

    // The fox's age.
    private int age;
    // Whether the fox is alive or not.
    private boolean alive;
    // The fox's position
    private Location location;
    // The fox's food level, which is increased by eating rabbits.
    private int foodLevel;
public Fox(boolean randomAge)
{
    age = 0; alive = true;
    if(randomAge) { age = rand.nextInt(MAX_AGE);
        foodLevel = rand.nextInt(RABBIT_FOOD_VALUE);
    }
    else { foodLevel = RABBIT_FOOD_VALUE; } // leave age at 0
}

public void hunt(Field currentField, Field updatedField, List newFoxes)
{
    incrementAge(); incrementHunger();
    if(isAlive()) {// New foxes are born into adjacent locations.
        int births = breed();
        for(int b = 0; b < births; b++) {
            Fox newFox = new Fox(false); newFoxes.add(newFox);
            Location loc = updatedField.randomAdjacentLocation(location);
            newFox.setLocation(loc); updatedField.place(newFox, loc); }
    // Move towards the source of food if found.
    Location newLocation = findFood(currentField, location);
    if(newLocation == null) { // no food found - move randomly
        newLocation = updatedField.freeAdjacentLocation(location); }
    if(newLocation != null) {
        setLocation(newLocation); updatedField.place(this, newLocation); }
    else { // can neither move nor stay - overcrowding
        alive = false; }
}
}
private void incrementAge()
{
    age++;
    if (age > MAX_AGE) { alive = false; }
}

private void incrementHunger()
{
    foodLevel--;
    if (foodLevel <= 0) { alive = false; }
}

private Location findFood(Field field, Location location)
{
    Iterator adjacentLocations = field.adjacentLocations(location);
    while (adjacentLocations.hasNext()) {
        Location where = (Location) adjacentLocations.next();
        Object animal = field.getObjectAt(where);
        if (animal instanceof Rabbit) {
            Rabbit rabbit = (Rabbit) animal;
            if (rabbit.isAlive()) {
                rabbit.setEaten();
                foodLevel = RABBIT_FOOD_VALUE;
                return where;
            }
        }
    }
    return null;
}
/**
 * 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) {
        births = rand.nextInt(MAX_LITTER_SIZE) + 1;
    }
    return births;
}

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

/**
 * Check whether the fox is alive or not.
 * @return True if the fox is still alive.
 */
public boolean isAlive()
{
    return alive;
}
public void setLocation(int row, int col)
{
    this.location = new Location(row, col);
}

public void setLocation(Location location)
{
    this.location = location;
}
Some Details about the Simulator

• The Simulator has three main activities
  – A constructor that sets up the field
  – The populate method that places the animals in the field; the initial populations are mixed-age
  – The simulateOneStep method that goes from the current step to the next step

• Several options are available to control the simulation
  – A run until finished mode
  – A run a fixed number of steps mode
  – A run for a single step mode
import java.util.Random;
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Collections;
import java.awt.Color;

public class Simulator {
  // The default width for the grid.
  private static final int DEFAULT_WIDTH = 50;
  // The default depth of the grid.
  private static final int DEFAULT_DEPTH = 50;
  // The probability that a fox will be created in any given grid position.
  private static final double FOX_CREATION_PROBABILITY = 0.02;
  // The probability that a rabbit will be created in any given grid position.
  private static final double RABBIT_CREATION_PROBABILITY = 0.08;
  // The list of animals in the field
  private List animals;
  // The list of animals just born
  private List newAnimals;
  // The current state of the field.
  private Field field;
  // A second field, used to build the next stage of the simulation.
  private Field updatedField;
  // The current step of the simulation.
  private int step;
  // A graphical view of the simulation.
  private SimulatorView view;
public Simulator()
{  this(DEFAULT_DEPTH, DEFAULT_WIDTH); }

/**
 * Create a simulation field with the given size.
 * @param depth Depth of the field. Must be greater than zero.
 * @param width Width of the field. Must be greater than zero.
 */
public Simulator(int depth, int width)
{
    if(depth <= 0 || depth <= 0) {
        System.out.println("The dimensions must be greater than zero.");
        System.out.println("Using default values.");
        depth = DEFAULT_DEPTH;  width = DEFAULT_WIDTH;
    }
    animals = new ArrayList();
    newAnimals = new ArrayList();
    field = new Field(depth, width);
    updatedField = new Field(depth, width);

    // Create a view of the state of each location in the field.
    view = new SimulatorView(depth, width);
    view.setColor(Fox.class, Color.blue);
    view.setColor(Rabbit.class, Color.orange);

    // Setup a valid starting point.
    reset();
}
public void runLongSimulation()
{
    simulate(500);
}

public void simulate(int numSteps)
{
    for(int step = 1; step <= numSteps && view.isViable(field); step++) {
        simulateOneStep();
    }
}

public void reset()
{
    step = 0;
    animals.clear();
    field.clear();
    updatedField.clear();
    populate(field);

    // Show the starting state in the view.
    view.showStatus(step, field);
}
public void simulateOneStep() {
    step++; newAnimals.clear();
    // let all animals act
    for(Iterator iter = animals.iterator(); iter.hasNext(); ) {
        Object animal = iter.next();
        if(animal instanceof Rabbit) {
            Rabbit rabbit = (Rabbit)animal;
            if(rabbit.isAlive()) {
                rabbit.run(updatedField, newAnimals);
            }
        } else if(animal instanceof Fox) {
            Fox fox = (Fox)animal;
            if(fox.isAlive()) {
                fox.hunt(field, updatedField, newAnimals);
            }
        } else { System.out.println("found unknown animal"); }
        iter.remove(); // remove dead animals from collection
    }
    // add new born animals to the list of animals
    animals.addAll(newAnimals);
    // Swap the field and updatedField at the end of the step.
    Field temp = field;
    field = updatedField;
    updatedField = temp;
    updatedField.clear(); // display the new field on screen
    view.showStatus(step, field);
}
/**
 * Populate the field with foxes and rabbits.
 */

private void populate(Field field)
{
    Random rand = new Random();
    field.clear();
    for(int row = 0; row < field.getDepth(); row++) {
        for(int col = 0; col < field.getWidth(); col++) {
            if(rand.nextDouble() <= FOX_CREATION_PROBABILITY) {
                Fox fox = new Fox(true);
                animals.add(fox);
                fox.setLocation(row, col);
                field.place(fox, row, col);
            }
            else if(rand.nextDouble() <= RABBIT_CREATION_PROBABILITY) {
                Rabbit rabbit = new Rabbit(true);
                animals.add(rabbit);
                rabbit.setLocation(row, col);
                field.place(rabbit, row, col);
            }
            // else leave the location empty.
        }
    }
    Collections.shuffle(animals);
}
Version 2 Uses Inheritance

- The fox and rabbit have come common values and behavior that can be abstracted to a parent class called Animal
- The common fields are age, alive and Location
- The common methods include getter/setter methods for these values
import java.util.List;

/**
 * Animal is an abstract superclass for animals. It provides features
 * common to all animals, such as the location and age.
 *
 * @author David J. Barnes and Michael Kolling
 * @version 2002-04-11
 */

public abstract class Animal
{
    // The animal's age.
    private int age;
    // Whether the animal is alive or not.
    private boolean alive;
    // The animal's position
    private Location location;

    /**
     * Create a new animal with age zero (a new born).
     */
    public Animal()
    {
        age = 0;
        alive = true;
    }
}
abstract public void act(Field currentField,
                        Field updatedField, List newAnimals);

public boolean isAlive()
{
    return alive;
}

public void setDead()
{
    alive = false;
}

public int getAge()
{
    return age;
}

public void setAge(int age)
{
    this.age = age;
}
/**
 * Return the animal's location.
 * @return The animal's location.
 */
public Location getLocation()
{
    return location;
}

/**
 * Set the animal's location.
 * @param row The vertical coordinate of the location.
 * @param col The horizontal coordinate of the location.
 */
public void setLocation(int row, int col)
{
    this.location = new Location(row, col);
}

/**
 * Set the animal's location.
 * @param location The animal's location.
 */
public void setLocation(Location location)
{
    this.location = location;
}
Improved Simulator

- The simulator behaves like before, but now it can treat all animals equally.
- What an animal does from step to step is defined in the method `act`; each type of animal overrides this method to act differently.

```java
// let all animals act
for(Iterator iter = animals.iterator(); iter.hasNext(); ) {
    Animal animal = (Animal)iter.next();
    if (animal.isAlive()) {
        animal.act(field, updatedField, newAnimals);
    } else {
        iter.remove();   // remove dead animals from collection
        // sorry, that's the rules of nature
    }
}
```
The Revised Rabbit Class

• Constants defining rabbit behavior remain local
• These include the breading age, the maximum age, the breeding probability, and the litter size
• When creating the original rabbits, we must introduce a random age factor to have a stable population
import java.util.List;
import java.util.Random;

/**
 * A simple model of a rabbit.
 * Rabbits age, move, breed, and die.
 *
 */

public class Rabbit extends Animal {

    // The age at which a rabbit can start to breed.
    private static final int BREEDING_AGE = 5;
    // The age to which a rabbit can live.
    private static final int MAX_AGE = 50;
    // The likelihood of a rabbit breeding.
    private static final double BREEDING_PROBABILITY = 0.15;
    // The maximum number of births.
    private static final int MAX_LITTER_SIZE = 5;
    // A shared random number generator to control breeding.
    private static final Random rand = new Random();

    public Rabbit(boolean randomAge) {
        super();
        if(randomAge) {
            setAge(rand.nextInt(MAX_AGE));
        }
    }
}
/**
 * This is what the rabbit does most of the time - it runs around. Sometimes it will breed or die of old age.
 */

public void act(Field currentField, Field updatedField, List newAnimals) {
    incrementAge();
    if(isAlive()) {
        int births = breed();
        for(int b = 0; b < births; b++) {
            Rabbit newRabbit = new Rabbit(false);
            newAnimals.add(newRabbit);
            newRabbit.setLocation(updatedField.randomAdjacentLocation(getLocation()));
            updatedField.place(newRabbit);
        }
        Location newLocation = updatedField.freeAdjacentLocation(getLocation());
        // Only transfer to the updated field if there was a free location
        if(newLocation != null) {
            setLocation(newLocation);
            updatedField.place(this);
        } else {
            // can neither move nor stay - overcrowding - all locations taken
            setDead();
        }
    }
}
private void incrementAge() {
    setAge(getAge() + 1);
    if (getAge() > MAX_AGE) {
        setDead();
    }
}

private int breed() {
    int births = 0;
    if (canBreed() && rand.nextDouble() <= BREEDING_PROBABILITY) {
        births = rand.nextInt(MAX_LITTER_SIZE) + 1;
    }
    return births;
}

public String toString() {
    return "Rabbit, age " + getAge();
}

private boolean canBreed() {
    return getAge() >= BREEDING_AGE;
}
import java.util.List;
import java.util.Iterator;
import java.util.Random;

/**
 * A simple model of a fox.
 * Foxes age, move, eat rabbits, and die.
 *
 * @author David J. Barnes and Michael Kolling
 * @version 2002-04-11
 */

public class Fox extends Animal {

    // The age at which a fox can start to breed.
    private static final int BREEDING_AGE = 10;
    // The age to which a fox can live.
    private static final int MAX_AGE = 150;
    // The likelihood of a fox breeding.
    private static final double BREEDING_PROBABILITY = 0.09;
    // The maximum number of births.
    private static final int MAX_LITTER_SIZE = 3;
    // The food value of a single rabbit. In effect, this is the
    // number of steps a fox can go before it has to eat again.
    private static final int RABBIT_FOOD_VALUE = 4;
    // A shared random number generator to control breeding.
    private static final Random rand = new Random();
    // The fox's food level, which is increased by eating rabbits.
    private int foodLevel;
public Fox(boolean randomAge)
{
    super();
    if(randomAge) {
        setAge(rand.nextInt(MAX_AGE));
        foodLevel = rand.nextInt(RABBIT_FOOD_VALUE);
    } else {
        // leave age at 0
        foodLevel = RABBIT_FOOD_VALUE;
    }
}

private void incrementAge()
{
    setAge(getAge() + 1);
    if(getAge() > MAX_AGE) {
        setDead();
    }
}

private void incrementHunger()
{
    foodLevel--;
    if(foodLevel <= 0) {
        setDead();
    }
}
public void act(Field currentField, Field updatedField, List newAnimals) {
    incrementAge();
    incrementHunger();
    if(isAlive()) {
        // New foxes are born into adjacent locations.
        int births = breed();
        for(int b = 0; b < births; b++) {
            Fox newFox = new Fox(false);
            newAnimals.add(newFox);
            newFox.setLocation(updatedField.randomAdjacentLocation(getLocation()));
            updatedField.place(newFox);
        }
        // Move towards the source of food if found.
        Location newLocation = findFood(currentField, getLocation());
        if(newLocation == null) {  // no food found - move randomly
            newLocation = updatedField.freeAdjacentLocation(getLocation());
        }
        if(newLocation != null) {
            setLocation(newLocation);
            updatedField.place(this);  // sets location
        } else {
            // can neither move nor stay - overcrowding - all locations taken
            setDead();
        }
    }
}
private Location findFood(Field field, Location location) {
    Iterator adjacentLocations =
        field.adjacentLocations(location);
    while(adjacentLocations.hasNext()) {
        Location where = (Location) adjacentLocations.next();
        Object animal = field.getObjectAt(where);
        if(animal instanceof Rabbit) {
            Rabbit rabbit = (Rabbit) animal;
            if(rabbit.isAlive()) {
                rabbit.setDead();
                foodLevel = RABBIT_FOOD_VALUE;
                return where;
            }
        }
    }
    return null;
}

private int breed() {
    int births = 0;
    if(canBreed() && rand.nextDouble() <= BREEDING_PROBABILITY) {
        births = rand.nextInt(MAX_LITTER_SIZE) + 1;
    }
    return births;
}
/**
 * @return A string representation of the fox.
 */
public String toString()
{
    return "Fox, age " + getAge();
}

/**
 * A fox can breed if it has reached the breeding age.
 */
private boolean canBreed()
{
    return getAge() >= BREEDING_AGE;
}
The Field Class

- A Field contains a two-dimensional array with one Animal in each cell or perhaps be empty.
- We need to be able to place Animals in the field or determine what animal exists at a particular location.
- This simulation is based on adjacent cells, for example a Fox can only eat a Rabbit if it is in an adjacent cell.
- This class returns an iterator over adjacent locations.
import java.util.Collections;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.Random;

/**
 * Represent a rectangular grid of field positions.
 * Each position is able to store a single animal.
 *
 * @author David J. Barnes and Michael Kolling
 * @version 2002-04-09
 */

public class Field
{
    private static final Random rand = new Random();

    // The depth and width of the field.
    private int depth, width;
    // Storage for the animals.
    private Animal[][] field;

    public Field(int depth, int width)
    {
        this.depth = depth;
        this.width = width;
        field = new Animal[depth][width];
    }
}
public void clear()
{
    for(int row = 0; row < depth; row++) {
        for(int col = 0; col < width; col++) {
            field[row][col] = null;
        }
    }
}

public void place(Animal animal)
{
    Location location = animal.getLocation();
    field[location.getRow()][location.getCol()] = animal;
}

public Animal getObjectAt(Location location)
{
    return getObjectAt(location.getRow(), location.getCol());
}

public Animal getObjectAt(int row, int col)
{
    return field[row][col];
}
public Location randomAdjacentLocation(Location location) {
    int row = location.getRow();
    int col = location.getCol();
    // Generate an offset of -1, 0, or +1 for both the current row and col.
    int nextRow = row + rand.nextInt(3) - 1;
    int nextCol = col + rand.nextInt(3) - 1;
    // Check in case the new location is outside the bounds.
    if(nextRow < 0 || nextRow >= depth || nextCol < 0 || nextCol >= width) {
        return location;
    } else if(nextRow != row || nextCol != col) {
        return new Location(nextRow, nextCol);
    }
    else { return location; }
}

public Location freeAdjacentLocation(Location location) {
    Iterator adjacent = adjacentLocations(location);
    while(adjacent.hasNext()) {
        Location next = (Location) adjacent.next();
        if(field[next.getRow()][next.getCol()] == null) {
            return next;
        }
    }
    // check whether current location is free
    if(field[location.getRow()][location.getCol()] == null) {
        return location;
    } else { return null; }
}
public Iterator adjacentLocations(Location location) {
    int row = location.getRow();
    int col = location.getCol();
    LinkedList locations = new LinkedList();
    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));
                }
            }
        }
    }
    Collections.shuffle(locations, rand);
    return locations.iterator();
}

public int getDepth() {
    return depth;
}

public int getWidth() {
    return width;
}
The Location Class

• This class stores the row and column values to locate a particular object in the field

• This class is interesting because it overrides three methods associated with all objects
  – It overrides the equals method; what would make two locations equal?
  – It overrides the toString method to print its value
  – It overrides the hashCode method to insure that every location possible in a field has a unique hash code
public class Location
{
    // Row and column positions.
    private int row;
    private int col;

    public Location(int row, int col)
    {
        this.row = row;
        this.col = col;
    }

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

    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.
 */
public int hashCode()
{
    return (row << 16) + col;
}

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

/**
 * @return The column.
 */
public int getCol()
{
    return col;
}
import java.awt.Color;
import java.util.HashMap;
import java.util.Iterator;

public class FieldStats {
    // Counters for each type of entity (fox, rabbit, etc.) in the simulation.
    private HashMap counters;
    // Whether the counters are currently up to date.
    private boolean countsValid;

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

    /**
     * Indicate that an animal count has been completed.
     */
    public void countFinished() {
        countsValid = true;
    }
}
public String getPopulationDetails(Field field) {
    StringBuffer buffer = new StringBuffer();
    if(!countsValid) {
        generateCounts(field);
    }
    Iterator keys = counters.keySet().iterator();
    while(keys.hasNext()) {
        Counter info = (Counter) counters.get(keys.next());
        buffer.append(info.getName());
        buffer.append(': ');
        buffer.append(info.getCount());
        buffer.append(' ');
    }
    return buffer.toString();
}

public void reset() {
    countsValid = false;
    Iterator keys = counters.keySet().iterator();
    while(keys.hasNext()) {
        Counter cnt = (Counter) counters.get(keys.next());
        cnt.reset();
    }
}
public void incrementCount(Class animalClass) {
    Counter cnt = (Counter) counters.get(animalClass);
    if(cnt == null) {
        // we do not have a counter for this species yet - create one
        cnt = new Counter(animalClass.getName());
        counters.put(animalClass, cnt);
    }
    cnt.increment();
}

public boolean isViable(Field field) {
    // How many counts are non-zero.
    int nonZero = 0;
    if(!countsValid) {
        generateCounts(field);
    }
    Iterator keys = counters.keySet().iterator();
    while(keys.hasNext()) {
        Counter info = (Counter) counters.get(keys.next());
        if(info.getCount() > 0) {
            nonZero++;
        }
    }
    return nonZero > 1;
}
/**
 * Generate counts of the number of foxes and rabbits. These are not kept up to date as foxes and rabbits are placed in the field, but only when a request is made for the information.
 */

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());
                // this tells us if it is a Fox or Rabbit
            }
        }
    }
    countsValid = true;
}
What We Have Learned

• We have explicitly dealt with the following topics:
  – A step-by-step simulation
  – Using a two dimensional array
  – The value of abstraction and inheritance
  – Designing for scalability
  – Determining the class of an object

• In the Lab you have learned
  – Adapting a design to an entirely new problem
  – More details on processing a two-dimensional array
  – How to handle limiting conditions, such as cells located on the border