Exercise 10.1

Yes, the number of foxes changes.

However, note that although this has a high probability of happening, it is not guaranteed to and will ultimately depend upon the initial state of the simulation.

Exercise 10.2

Yes, it changes each time simulateOneStep is invoked.

Sometimes the number increase and other times it decreases. It probably simulates the births and deaths of foxes.

Once again, note that it is highly probably that it will change but there are some states of the simulation when it will not change over the course of a single step.

Exercise 10.3

No.

Exercise 10.4

The numbers of foxes and rabbits goes up and down. The simulation will stop before the end of the specified number of steps if either animal dies out completely.

Exercise 10.5

No, it is not an identical simulation.

Yes, it is similar patterns.

Exercise 10.6

Yes, sometimes the foxes or rabbits die out completely. If there are a lot of foxes they may eat all the rabbits. If there are not many rabbits then the foxes may die of hunger.

Exercise 10.12

Making the breeding probability of the rabbits much lower has the result that the foxes die out quickly because they don't have enough food. Increasing the breeding probability of the rabbits make them spread faster.

Exercise 10.11

Increasing the maximum age for foxes doesn't seem to have a big impact on the number of foxes.

Exercise 10.15

Yes, in some configurations the foxes or the rabbits disappear completely while in others a balance seems to exist.

Exercise 10.16

Yes the size does affect the likelihood of species surviving. With a small field (10x10) this is easy to observe.

Exercise 10.17

The modified findFood() method:

```
private Location findFood(Location location)
{
    List<Location> adjacent = field.adjacentLocations(location);
    Iterator<Location> it = adjacent.iterator();
    Location rabbitLocation = null;
    while(it.hasNext()) {
        Location where = it.next();
        Object animal = field.getObjectAt(where);
        if(animal instanceof Rabbit) {
            Rabbit rabbit = (Rabbit) animal;
            if(rabbit.isAlive()) {
                rabbit.setDead();
                foodLevel = RABBIT_FOOD_VALUE;
                rabbitLocation = where;
            }
        }
    }
    return rabbitLocation;
}
```

Exercise 10.18

To implement the suggested changes we need a field with maximum food value:

```
private static final int MAX_FOOD_VALUE = 20;
```

And replace the statement:

```
foodLevel = RABBIT_FOOD_VALUE;
```

with the statements:

```
foodLevel += RABBIT_FOOD_VALUE;
if(foodLevel > MAX_FOOD_VALUE) {
    foodLevel = MAX_FOOD_VALUE;
}
```

Depending on the maximum food value, the foxes seem to survive longer.

Exercise 10.19

It appears to be less likely that a balance is established if all animals start at age zero.

Exercise 10.20

It still appears to be difficult to establish a working balanced relationship.

The relative size of the initial populations doesn't have a big impact on the outcome of the simulation.

Exercise 10.22

The advantage would be avoiding any possible inconsistency between the content of the Field and the lists of animals.

The disadvantage is that it takes time to generate the lists.

Exercise 10.23

There are dead rabbits in the rabbit list that do not appear in the field. This is because the hunt method in fox is able to remove eat rabbits from the field, but it does not have access to the rabbit list in order to remove them.

Exercise 10.24

The similar class fields are:

```
    private static final int BREEDING_AGE;
    private static final int MAX_AGE;
    private static final double BREEDING_PROBABILITY;
    private static final int MAX_LITTER_SIZE;
    private static final Random rand = new Random();
```

The fox has an additional class field:

```
    private static final int RABBIT_FOOD_VALUE;
```

The similar instance fields are:

```
    private int age;
    private boolean alive;
    private Location location;
```

The fox has an additional instance field:

```
    private int foodLevel;
```

The constructors are similar, except that the Fox constructor also initialises its foodLevel.

The similar methods are:

```
    private void incrementAge() //except for the values of the static
fields
    private int breed() //except for the values of the static fields
    private boolean canBreed() //except for the values of the static
fields
    public Location getLocation()
    public boolean isAlive()
    private void setDead()
    public void setLocation(Location location)
```

The Rabbit class has these methods that the Fox class doesn't have:

```
    private void giveBirth(List<Rabbit> newRabbits)
    public void run(List<Rabbit> newRabbits)
```

And the Fox class has these methods that the Rabbit class doesn't have:

```
    private void giveBirth(List<Fox> newFoxes)
    public void hunt(List<Fox> newFoxes)
    private void incrementHunger()
    private Location findFood(Location location)
```

Exercise 10.25

The truly identical methods are:

```
    public Location getLocation()
    public boolean isAlive()
    private void setDead()
    public void setLocation(Location location)
```

Exercise 10.24

In this case, no. Because it is most likely that we would want to change the value in the future to values different for the Fox and the Rabbit.

In general it depends on the actual field. If it can truly be considered a field which will always have the same value for both the Rabbit and the Fox, it would make sense to treat methods that use that field as identical.

Exercise 10.27

A good way to test it would be to build a set of unit tests that tests the part of the program that is likely to be affected by the changes. This could be done by creating JUnit test classes for Fox and Rabbit.

After each small change to the program we should run the unit tests to see if the program still behaves as expected.

Running the program is also a good way to test this program, because it is easy to spot serious errors from the visual output of the program.

Exercise 10.28

See the project foxes-and-rabbits-v2 from the CD.

Exercise 10.29

We have avoided code duplication.

The classes Fox and Rabbit are smaller.

If we want to add new animals in the future, we already have some functionality available in the animal class.

Exercise 10.30

We can't treat the Animals as Objects because the object does not define the act() method that we need to use.

Exercise 10.31

Yes, a class must be declared abstract if it has abstract methods.

Exercise 10.32

Yes, a class can be declared as abstract even though it does not contain any abstract methods.

Exercise 10.33

If you want to prohibit instantiation of a class you could declare it abstract.

If you know that the class will never be instantiated, declaring it abstract would help other programmers in understanding the code

Exercise 10.34

AbstractCollection, AbstractSet, AbstractMap, AbstractList, AbstractSequentialList.

You can see that a class is abstract in the documentation. The first line below the heading says something like:

```
public abstract class AbstractCollection
```

To see which concrete classes extend them, take a look at the class diagram in the solution to exercise 8.15.

Exercise 10.35

Yes, you can see that a method is abstract in the API documentation. For instance, in the left column of the Method Summary.

We need to know this if we want to extend an abstract class with a concrete class. This tells us which methods that we as a minimum must implement.

Exercise 10.36

Because we use abstract methods in the Animal class which are overridden in the two subclasses Rabbit and Fox. To understand this behavior it is necessary to understand method overriding.

Exercise 10.37

Download: 10-37-foxes-and-rabbits.zip

Exercise 10.40

We need to put in the definition of the getBreedingAge in the Animal class:

```
protected abstract int getBreedingAge();
```

An implementation of this can be downloaded in the next exercise.

Exercise 10.41

Download: 10-41-foxes-and-rabbits.zip

Exercise 10.42

Yes, the breed() can be moved to the Animal class. We then also have to create methods to access the two static fields: BREEDING_PROBABILITY and MAX_LITTER_SIZE. Just as we did in the two previous exercises.

Exercise 10.44

The changes we have made to the Animal, Rabbit and Fox classes did not require us to modify other classes except the Simulator class (disregarding exercise 10.37 where we created the PopulationGenerator). This tells us that the original program had a low degree of coupling and good encapsulation.

Exercise 10.45

Download: 10-45-foxes-and-rabbits.zip

Exercise 10.46

Yes it is possible.

Define an abstract method in Animal that returns a new born Animal:

```
    /**
     * Create a new animal. An animal may be created with age
     * zero (a new born) or with a random age.
     *
     * @param randomAge If true, the animal will have a random age.
     * @param field The field currently occupied.
     * @param location The location within the field.
     */
    abstract protected Animal createAnimal(boolean randomAge,
                                           Field field, Location
location);
```

This makes it possible to move giveBirth from Fox and Rabbit to Animal as follows:

```
    /**
     * Check whether or not this animal is to give birth at this
step.
     * New births will be made into free adjacent locations.
     * @param newborn A list to add newly born animals to.
     */
    protected void giveBirth(List<Animal> newborn)
    {
        // New rabbits are born into adjacent locations.
        // Get a list of adjacent free locations.
        Field field = getField();
        List<Location> free =
field.getFreeAdjacentLocations(getLocation());
        int births = breed();
        for(int b = 0; b < births && free.size() > 0; b++) {
            Location loc = free.remove(0);
            newborn.add(createAnimal(false, field, loc));
        }
    }
```

Note that this calls createAnimal rather than creating either a Fox or a Rabbit directly. Fox and Rabbit define concrete versions of createAnimal, such as:

```
    protected Animal createAnimal(boolean randomAge, Field field,
Location location)
    {
        return new Rabbit(randomAge, field, location);
    }
```

Exercise 10.47

Need to add the import statement of java.util.List to the Actor class - but that is all.

Of course, we also need to specify that Animal extends Actor.

Furthermore, we should update the text of the Field class (variable names and comments) to reflect the use of Actors instead of Animals.

Exercise 10.48

Classes that extend Actor will now have to implement it instead.

Exercise 10.49

The fields are static and public fields. Interfaces only allow public static final fields.

Exercise 10.50

There are three errors in this program:

- The field THRESHOLD is declared private which is not allowed.
- It is not allowed to have constructors in interfaces.
- It has an implementation of getThreshold() - no implementations allowed in interfaces.

Exercise 10.51

Download: 10-51-foxes-and-rabbits.zip

The number of hunters varies. This is because the newly born animals just get placed into the field without checking whether an actor already occupies that field.

Only a few changes is necessary in the other classes:
- A new method needs to be introduced in the Field class: getRandomLocation()
- The populate method needs to be updated to also create hunters.

Exercise 10.52

```
ArrayList:
                          ensureCapacity
                          removeRange
                          trimToSize
LinkedList:
                          addFirst
                          addLast
                          getFirst
                          getLast
                          removeFirst
                          removeLast
```

The reason that these methods are not defined in the List interface is that the methods are not common to all lists, but are specific for that type of list implementation.

Exercise 10.53

The following interfaces are mentioned:
- List
- Comparator
- Comparable

Exercise 10.54

An implementation of a class implementing the Comparable interface:

```
/**
 * A class representing some kind of coffee.
 */
public class Coffee implements Comparable<Coffee>
{
    // The strength of the coffee
    private int strength;

    /**
     * Create a new coffee with the given strength
     */
    public Coffee(int strength)
    {
        this.strength = strength;
    }

    public int compareTo(Coffee other)
    {
        int otherStrength = other.strength;
        if (strength > otherStrength) {
            return 1;
        }
        else if (strength < otherStrength) {
            return -1;
        }
        else {
            return 0;
        }
    }

    public String toString()
    {
        return "" + strength;
    }
}
```

And a class to test that it is sorted correctly:

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;
public class Test
{
    public void testComparable()
    {
        List<Coffee> coffees = new ArrayList<Coffee>();
        coffees.add(new Coffee(10));
        coffees.add(new Coffee(2));
        coffees.add(new Coffee(10));
        coffees.add(new Coffee(20));
        coffees.add(new Coffee(5));
        Collections.sort(coffees);
        Iterator<Coffee> iter = coffees.iterator();
        while (iter.hasNext()) {
            System.out.println(iter.next());
```

```
        }
    }
}
```

Exercise 10.55

Download: 10-55-foxes-and-rabbits.zip

Exercise 10.56

Download: 10-56-foxes-and-rabbits.zip

Exercise 10.57

In the program below, we have created a list of views that all get updated when required.

This has been done in a way that allows any number of views to be added. To avoid making a lot of changes to the Simulator class, we have created a new kind of view (named MultiView) that can delegate the method calls to several other views.

Download: 10-57-foxes-and-rabbits.zip

Exercise 10.58

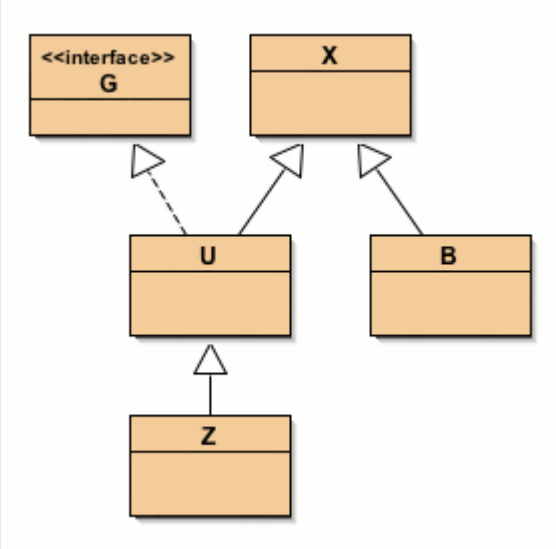
a)
Yes, an abstract class can have concrete methods. That is just the way it is - if it couldn't have concrete methods it would almost be the same as a Java interface.

b)
No, a concrete class can not have abstract methods. If it could, it would not be possible to instantiate an object of that class. What implementation should be run when you try to invoke an abstract method that is not implemented?

c)
Yes, you can have an abstract class without any abstract methods. See exercise 10.29.

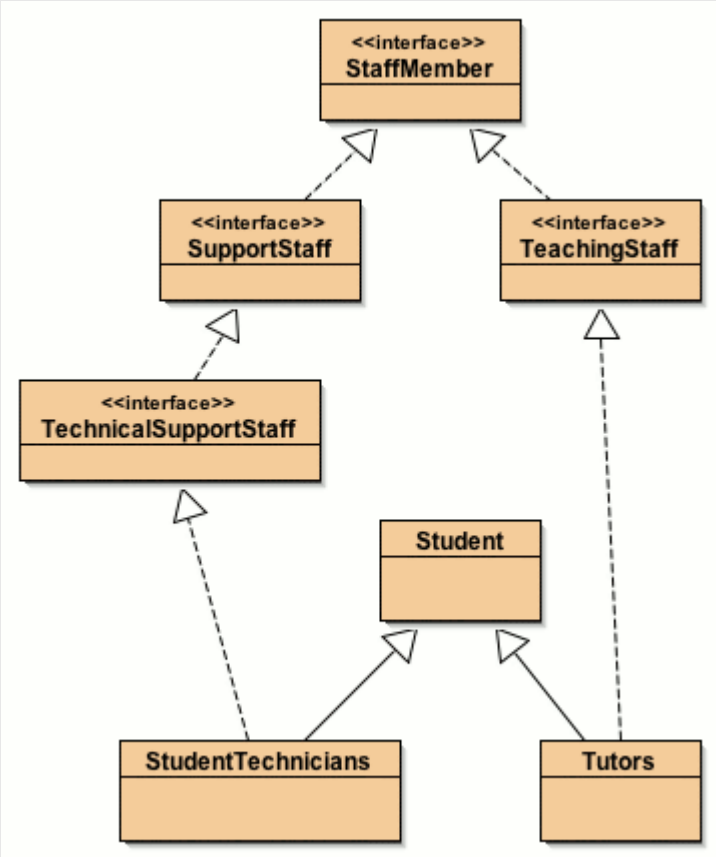
Exercise 10.59

All the types could be interfaces. Because G and X both are super classes for U (legal: g=u,x=u), and do not have any relationship between them (illegal: g=x, x=g), at least one of G or X must be an interface.



Exercise 10.60

Several possible hierarchies could be created. This is one example:



Exercise 10.61

The reason for the Adapter classes are only convenience for the programmer. If the programmer knows that only a few of the methods from an interface are going to be used, it is still necessary to write an implementation for all the methods in the interface. If the Adapter class is used instead, the programmer can just override the

few methods that are needed.

A common use of adapter classes is for instance the MouseAdapter. The MouseListener interface contains five methods to listen for mouse events. If you only want to listen for events when the mouse is clicked you would have to write something like this with the interface:

```
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;

public class ClickPrinter implements MouseListener
{
    public void mouseClicked(MouseEvent e)
    {
         System.out.println("Mouse clicked");
    }

    public void mousePressed(MouseEvent e)
    {
    }

    public void mouseReleased(MouseEvent e)
    {
    }

    public void mouseEntered(MouseEvent e)
    {
    }

    public void mouseExited(MouseEvent e)
    {
    }
}
```

And with the MouseAdapter it would look like this:

```
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

public class Main extends MouseAdapter
{
    public void mouseClicked(MouseEvent e)
    {
        System.out.println("Mouse clicked");
    }
}
```

Exercise 10.62

We need to give the TreeSet someway of knowing how to sort the elements. This can be done in two ways. Either we let Person implement the Comparable interface or we create a new class which extends Comparator and knows how to compare Persons. In exercise 10.54 we did something similar where we used the Comparable interface. So

we will try it out with the Comparator this time.

The Person class:

```
public class Person
{
    private int age;

    public Person(int age)
    {
        this.age = age;
    }

    public int getAge()
    {
        return age;
    }

    public boolean equals(Object other)
    {
        if(other instanceof Person) {
            return this.age == ((Person) other).age;
        }
        else {
            return false;
        }
    }

    public String toString()
    {
        return "" + age;
    }
}
```

The PersonComparator:

```
import java.util.Comparator;

public class PersonComparator implements Comparator<Person>
{
    public int compare(Person p1, Person p2)
    {
        if(p1.getAge() < p2.getAge()) {
            return -1;
        }
        else if(p1.getAge() == p2.getAge()) {
            return 0;
        }
        else { //(p1.getAge() > p2.getAge())
            return 1;
        }
    }
}
```

A test class:

```
import java.util.Set;
import java.util.TreeSet;
```

```
import java.util.Iterator;

public class Test
{
    public static void runTest()
    {
        Set<Person> persons = new TreeSet<Person>(new
PersonComparator());
        persons.add(new Person(32));
        persons.add(new Person(17));
        persons.add(new Person(13));
        persons.add(new Person(35));
        persons.add(new Person(27));
        Iterator<Person> iter = persons.iterator();
        while (iter.hasNext()) {
            System.out.println(iter.next());
        }
    }
}
```