|  |
| --- |
| **Exercise 14.1**  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 14.2**  Yes, in general it changes each time simulateOneStep() is invoked.  Sometimes the number increase and other times it decreases. It probably simulates the birth and death of foxes.  It is highly probably that it will change on each step, but there are some states of the simulation when it will not change over the course of a single step. For instance, when numbers are low. |
| **Exercise 14.3**  No, the rates vary. |
| **Exercise 14.4**  The numbers of foxes and rabbits go up and down. The simulation will stop before the end of the specified number of steps if either animal dies out completely. |
| **Exercise 14.5**  No, it is not an identical simulation.  Yes, it exhibits similar patterns. |
| **Exercise 14.6**  Yes, sometimes the foxes or rabbits die out completely. If there are many foxes near the rabbit groupings then they may eat all the rabbits. If there are not many rabbits, or the foxes become isolated from them, then the foxes may die of hunger. |
| **Exercise 14.9**  **The** reset() **method of the** Randomizer **must be called first, because this affects the creation of the initial populations following a reset in the** Simulator**.** |
| **Exercise 14.11**  **Distinguishing between sex is only useful if a more accurate simulation of breeding is included in the model. Given the other simplifications in the model, it is unlikely that more accurate breeding patterns would add in a significant way to the lessons learned from the model. Similar effects on numbers are likely to be achievable through other, simpler modifications to existing parameters – such as the breeding frequency or litter sizes.** |
| **Exercise 14.12**  **One observation is that the 'environment' is treated as completely uniform – any animal can occupy any part of it. A real environment would be far less uniform than this.**  **Feeding has also been treated simplistically, with their being no resource constraints on the rabbits, for instance. In practice, a very large population of rabbits would eat themselves out of their food supply.** |
| **Exercise 14.13**  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 14.15**  Increasing the maximum age for foxes doesn't seem to have a big impact on the number of foxes, suggesting that their survival is largely determined by available food rather than age. |
| **Exercise 14.16**  Yes, in some configurations the foxes or the rabbits disappear completely while in others a balance seems to exist. |
| **Exercise 14.17**  Yes, the size does affect the likelihood of species surviving. With a small field (30x30, say) this is easy to observe. |
| **Exercise 14.19**  **As Exercise 14.17 illustrates, an uneven division resulting in too small an area for one of the partitions will likely lead to extinction of one of the species.** |
| **Exercise 14.20**  It is only necessary to change the loop condition in findFood from the following:  while(foodLocation == null && it.hasNext())  to:  while(it.hasNext()) |
| **Exercise 14.21**  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 14.22**  **The normal fluctuations in numbers often mean that the absolute number of individuals for a particular species drops very low from time to time – a single-digit figure, for instance. It only requires a run of 'bad luck' – i.e., random variation – such as the few remaining foxes heading away from the rabbits instead of towards them, or breeding in low numbers for a few cycles – to take the population over the cliff-edge of extinction. While this might have a low probability in normal circumstances, the probability will never be zero.**  **It might be worth adding monitoring output to the simulation to show just how low the population levels become, from time to time.** |
| **Exercise 14.23**  It appears to be less likely that a balance is established if all animals start at age zero. |
| **Exercise 14.24**  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 14.25**  **There are dead rabbits that do not appear in the field. This is because the** findFood **method in Fox eat rabbits but it does not remove them from either copy of the field.** |
| **Exercise 14.26**  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 = Randomizer.getRandom();  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;  private Field field;  The fox has an additional instance field:  private int foodLevel;  The constructors are similar, except that the Fox constructor also initializes its food level.  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()  private void giveBirth(Field nextFieldState,  List<Location> freeLocations)  public boolean isAlive()  public void setDead()  public String toString()  The Rabbit class has this method that the Fox class doesn't have:  public void run(Field currentField, Field nextFieldState)  And the Fox class has these methods that the Rabbit class doesn't have:  public void hunt(Field currentField, Field nextFieldState)  private void incrementHunger()  private Location findFood(Field field) |
| **Exercise 14.27**  The truly identical methods are:  public Location getLocation()  public boolean isAlive()  private void setDead() |
| **Exercise 14.28**  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 attribute of the animals being modelled. If it can truly be considered that an attribute will always have the same value for both the Rabbit and the Fox, then it would make sense to treat methods that use that field as identical. |
| **Exercise 14.29**  A good way to test it would be to build a set of unit tests that test the parts of the program that are 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. As noted in the text of Exercise 14.30, being able to control the random elements of the simulation via the Randomizer class will help with some aspects of this. However, at some point, the changes are likely to affect the way in which the random values are used and, at that point, the two versions will naturally diverge.  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 14.30**  See the project foxes-and-rabbits-v2. |
| **Exercise 14.31**  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 14.32**  We can't treat the Animal as Object because Object does not define an act() method that we need to use. |
| **Exercise 14.33**  Yes, a class must be declared abstract if it has abstract methods. |
| **Exercise 14.34**  Yes, a class can be declared as abstract even though it does not contain any abstract methods. |
| **Exercise 14.35**  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 14.36**  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 12.15. |
| **Exercise 14.37**  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 14.38**  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 14.39**  See: 14-39-foxes-and-rabbits |
| **Exercise 14.40**  **The graph shows the rise and fall of population numbers of the two species over time. Observe how rises in the fox population tend to follow slightly behind rises in the rabbit population, and then lead on to falls in rabbit numbers, and consequent falls in fox numbers. These linked cycles clearly show how closely related the population numbers are.** |
| **Exercise 14.44**  We need to put in the definition of the abstract getBreedingAge() in the Animal class:  protected abstract int getBreedingAge();  An implementation of this can be found in the next exercise. |
| **Exercise 14.45**  See: 14-45-foxes-and-rabbits |
| **Exercise 14.46**  Yes, 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 14.48**  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 14.41 where we created the PopulationGenerator). This tells us that the original program had a low degree of coupling and good encapsulation. |
| **Exercise 14.49**  See: 14-49-foxes-and-rabbits |
| **Exercise 14.50**  **Yes, it is possible.** See: 14-50-foxes-and-rabbits  **Define an abstract method in** Animal **that returns a newborn** 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,  Location location);  **This makes it possible to move** giveBirth() **from** Fox **and** Rabbit **to** Animal.  **The method now 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,  Location location)  {  return new Rabbit(randomAge, field, location);  }  Also moved is the breed method, along with defining abstract methods in Animal for the breeding probability and litter size, plus the concrete versions of those in the Fox and Rabbit classes. |
| **Exercise 14.51**  See: 14-51-foxes-and-rabbits  We need to specify that Animal extends Actor and the act method can now be removed from Animal.  Either the isAlive method in Animal needs to be renamed to isActive, or isActive needs to be added to call isAlive.  Furthermore, we should update the Field class (method names, variable names and comments) to reflect the use of Actor instead of Animal. That will have a knock-on effect on the other classes. |
| **Exercise 14.52**  Classes that extend Actor will now have to implement it instead. |
| **Exercise 14.53**  See: 14-53-foxes-and-rabbits  The number of scientists reduces over time. This is because the newly born animals just get placed into the next field state without checking whether an actor already occupies that location. Sometimes, a scientist is 'over written' in this way. Because scientists are only created at the start of the simulation, their number never increases.  Only a few changes are 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 create scientists. - A color needs to be defined in the SimulatorView. |
| **Exercise 14.54**   |  |  | | --- | --- | | 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 14.56**  The following interfaces are mentioned: List, Comparator, Comparable.  replaceAll, sort and spliterator have default implementations in the List interface. |
| **Exercise 14.57**  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;  }  /\*\*  \* Compare the strength of this coffee with  \* the other coffee.  \* @param other The other coffee.  \* @return the result of the comparison.  \*/  public int compareTo(Coffee other)  {  return strength - other.strength;  }    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<>();  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);    System.out.println("Coffees in order of strength:");  for(Coffee type : coffees) {  System.out.println(type);  }  } } |
| **Exercise 14.59**  See: 14-59-foxes-and-rabbits |
| **Exercise 14.60**  See: 14-60-foxes-and-rabbits  In effect, this is what would be done in Exercise 14.59 if TextView is added to the list of views, along with GridView and GraphView, instead of replacing GridView in the list. |
| **Exercise 14.62**  **The** PriorityQueue **class might be useful for ordering events.** |
| **Exercise 14.64**  The fields are static and public fields. Interfaces only allow public static final fields. |
| **Exercise 14.65**  There are the following errors in this interface:   * The fields THRESHOLD and value are declared private which is not allowed. They must be public. * The field value needs to be initialized explicitly. * It is not allowed to have constructors in interfaces. * The implementation of getThreshold() must be declared as default if it has an implementation. |
| **Exercise 14.66**  **(This is a consolidation of earlier exercises.)**  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. |
| **Exercise 14.67**  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 14.68**  Several possible hierarchies could be created. This is one example: |
| **Exercise 14.69**  The reason for the Adapter classes is a convenience for programmers. If a 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, a 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 14.70**  We need to give the TreeSet some way 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 Person objects. In Exercise 14.57 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 otherPerson) {  return this.age == otherPerson.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)  {  return p1.getAge() - p2.getAge();  } }  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<>(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());  }  }  } |