Further Abstraction Techniques

Java version

Objectives

This section completes Java's inheritance mechanism with more advanced techniques: abstract classes and interfaces.

Main concepts discussed in this chapter

- abstract classes
- interfaces

Resources

- Classes needed for this lab - *chapter12.jar*. Download and install as usual.

To do

Simulations

Computer simulations try to model aspects of the real world by computer. There is of course a trade-off: the more realistic the simulation, the more complex the program and the more computer power is needed.

Our simulation will necessarily be simpler than the scenario we have described, because we are using it mainly to illustrate new features of object-oriented design and implementation. Therefore, it will not have the potential to simulate accurately many aspects of nature, but some of the simulation’s characteristics are nonetheless interesting. In particular, it will demonstrate the structure of typical simulations. In addition, its accuracy may surprise you; it would be a mistake to equate greater complexity with greater accuracy. It is often the case that a simplified model of something can provide greater insight and understanding than a more complex one, from which it is often difficult to isolate the underlying mechanisms—or even be sure that the model is valid.

The foxes-and-rabbits simulation

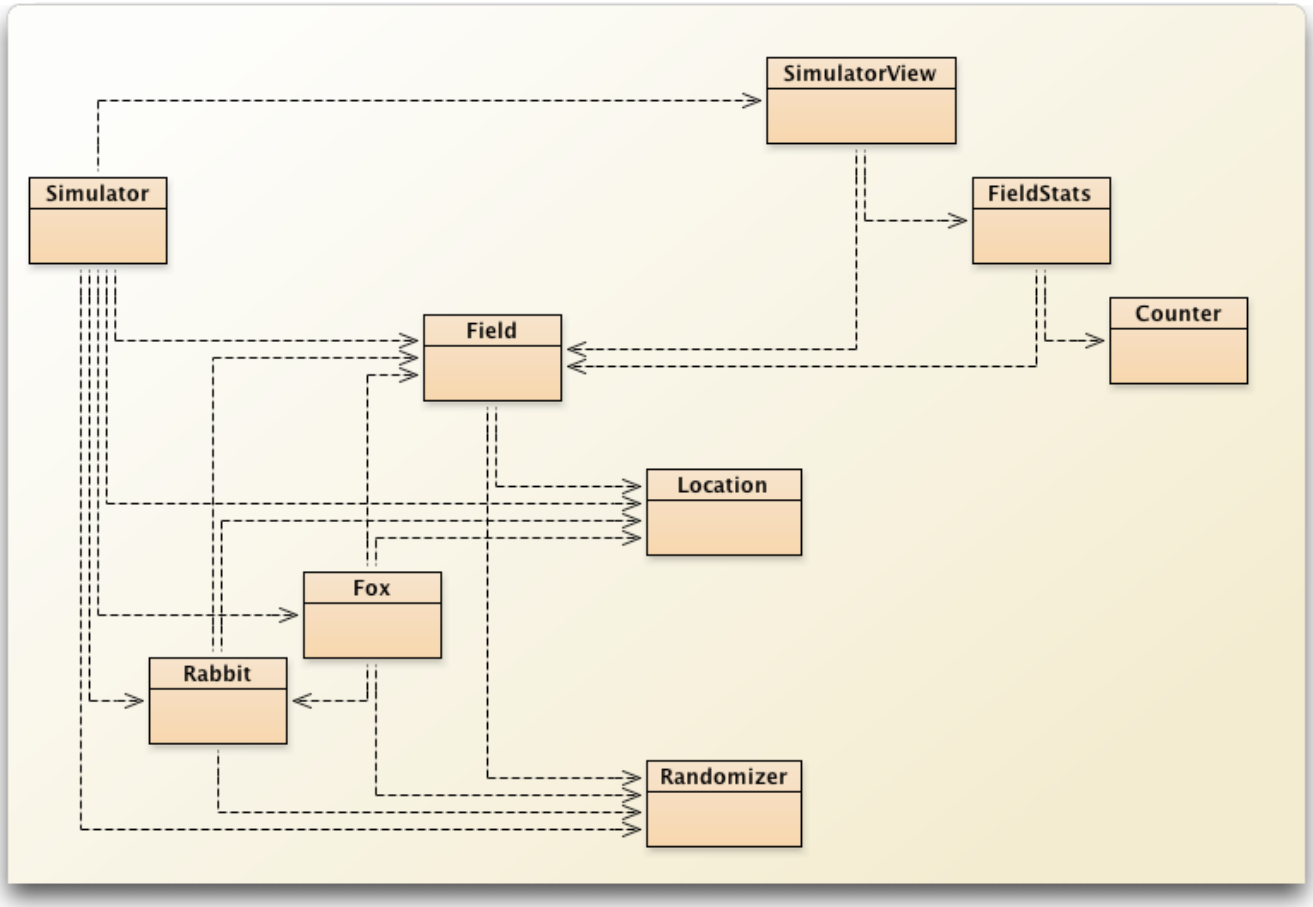
Environmental impact studies often involve computer simulations, *what happens if we do this?* Population dynamics can involve *predator-prey* simulations where two species share an environment and one species preys on the other. This is classically modeled mathematically by the *Volterra predator-prey* nonlinear differential equations.

The simulation scenario we have chosen to work with in this chapter involves tracking populations of foxes and rabbits within an enclosed area. This is just one particular example of predator–prey simulations. Such simulations are often used to model the variation in population sizes that result from a predator species feeding on a prey species. A delicate balance exists between such species. A large population of prey will potentially provide plenty of food for a small population of predators. However, too many predators could kill off all the prey and leave the hunters with nothing to eat. Population sizes could also be affected by the size and nature of the environment. For instance, a small, enclosed environment could lead to overcrowding and make it easy for the predators to locate their prey, or a polluted environment could reduce the stock of prey and prevent even a modest population of predators from surviving. Because predators in one context are often prey for other species (think of cats, birds, and worms, for instance), loss of one part of the food chain can have dramatic effects on the survival of other parts.

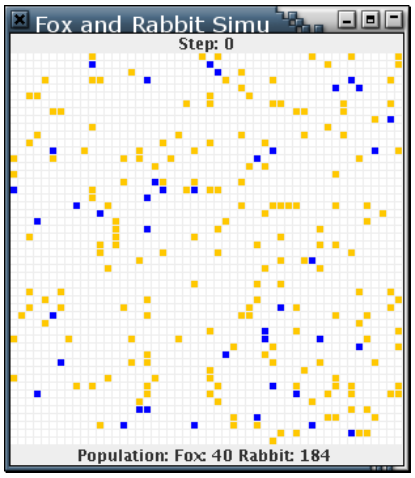
We again start with a poor working version, which we'll improve by introducing further abstraction techniques.

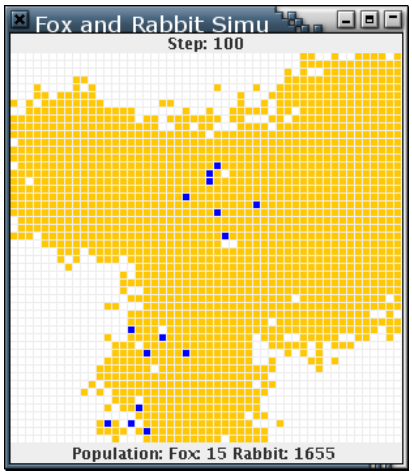
The foxes-and-rabbits project

Open the *foxesandrabbits.v1* project. The class diagram looks like



The **`Simulator`** class holds a collection of **`Fox`**es and **`Rabbit`**s. At each *step* of the **`Simulator`**, each fox and rabbit is allowed to move. After each step, the state of the environment is displayed by the **`SimulatorView`**:





Exercises

1. You will need a **`Main`** class to run the simulation. Create a **`Simulator`** object using the no-args constructor. Does the number of foxes (the less numerous squares) change if you call the **`simulateOneStep`** method just once?
2. Does the number of foxes change on every step? What natural processes do you think we are modeling that cause the number of foxes to increase or decrease?

3. Call the **simulate** method to run the simulation continuously for a significant number of steps, such as 50 or 100. Do the numbers of foxes and rabbits increase or decrease at similar rates?
4. What change do you notice if you run the simulation for a very long time, eg, 500 steps. Use the **runLongSimulation** method.
5. Use the **reset** method to restore the starting state of the simulation and then run it again. Is an identical simulation run this time? If not, do you see broadly similar patterns emerging anyway? Does the simulation behaviour have any relation to *Volterra's equations*?
6. If you run the simulations long enough, do all of the foxes or all of the rabbits ever die off completely? If so, can you pinpoint any reasons why that might be occurring?
7. In the source code of the **Simulator** class, find the **simulate** method. In its body, you will see a call to a **delay** method that is commented out. Uncomment this call and run the simulation. Experiment with different delays so you can observe the simulation behavior more clearly. At the end, leave it in a state that makes the simulation look useful on your computer.
8. Note the number of rabbits and foxes at the start, after a few steps, and at the end of a long simulation. This can be useful when you modify the code and observe what happens, as well as for testing.
9. After running the simulation, re-initialize and call the static method **Randomizer.reset**. Run the simulation again - this *should* give the same results each time. If it doesn't, look at the code in the **Randomizer** class and try to find the problem. You may have to look at the API documentation for **java.util.Random**.
10. Check the effect of changing the value of the attribute **Randomizer.useShared**. Be sure to restore it to **true** afterwards, because repeatability will be an important element in later testing.

The Rabbit class

```
Import statements and class comment omitted.

class Rabbit {
    // Characteristics shared by all rabbits (class variables).

    // 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 = 40;
    // The likelihood of a rabbit breeding.
    private static final double BREEDING_PROBABILITY = 0.12;
    // The maximum number of births.
    private static final int MAX_LITTER_SIZE = 4;
    // A shared random number generator to control breeding.
    private static final Random rand = Randomizer.getRandom();

    // 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;
// The field occupied.
private Field field;

/**
 * Create a new rabbit. A rabbit may be created with age zero
 * (a new born) or with a random age.
 *
 * @param randomAge
 *            If true, the rabbit will have a random age.
 * @param field
 *            The field currently occupied.
 * @param location
 *            The location within the field.
 */
Rabbit(boolean randomAge, Field field, Location location) {
    // body of constructor omitted
}

/**
 * This is what the rabbit does most of the time - it runs around.
 * Sometimes it will breed or die of old age.
 *
 * @param newRabbits
 *            A list to return newly born rabbits.
 */
void run(List<Rabbit> newRabbits) {
    incrementAge();
    if (alive) {
        giveBirth(newRabbits);
        // Try to move into a free location.
        Location newLocation = field.freeAdjacentLocation(location);
        if (newLocation != null) {
            setLocation(newLocation);
        } else {
            // Overcrowding.
            setDead();
        }
    }
}

/**
 * Indicate that the rabbit is no longer alive. It is removed from
 * the field.
 */
void setDead() {
    alive = false;
    if (location != null) {
        field.clear(location);
        location = null;
        field = null;
    }
}
```

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

    /**
     * Check whether or not this rabbit is to give birth at this step.
     * New births will be made into free adjacent locations.
     *
     * @param newRabbits
     *            A list to return newly born rabbits.
     */
    private void giveBirth(List<Rabbit> newRabbits) {
        // New rabbits are born into adjacent locations.
        // Get a list of adjacent free locations.
        List<Location> free = field.getFreeAdjacentLocations(location);
        int births = breed();
        for (int b = 0; b < births && free.size() > 0; b++) {
            Location loc = free.remove(0);
            Rabbit young = new Rabbit(false, field, loc);
            newRabbits.add(young);
        }
    }

    /**
     * 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;
    }

    Other methods omitted.
}
```

Rabbits have several constants (static final variables), common to all rabbits, and instance variables to describe the state of each individual rabbit. A rabbit's behaviour is controlled by its `run` method. A rabbit can

- move (randomly),

- grow older,
- and breed (randomly) if it's old enough.

Exercises

11. Do you think that having all rabbits being capable of breeding will produce unrealistic simulations?
12. What simplifications are made by the **Rabbit** class compared to real rabbits? Do you feel that these could have a significant impact on the accuracy of the simulation?
13. Experiment with the effects of altering some or all of the values of the static variables in the **Rabbit** class. For instance, what effect does it have on the fox and rabbit populations if the breeding probability of rabbits is much higher or lower?

The Fox class

The **Fox** class is quite similar to **Rabbit**, here are only the things which are distinctive to foxes

```
Import statements and class comment omitted.

class Fox {
    // Characteristics shared by all foxes (class variables).

    // 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 = 9;

    // other static fields omittee

    // Individual characteristics (instance fields).

    // 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 field occupied.
    private Field field;
    // The fox's food level, which is increased by eating rabbits.
    private int foodLevel;

    /**
     * Create a fox. A fox 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 fox will have random age and hunger level.
     * @param field
     *             The field currently occupied.
     * @param location
     *             The location within the field.
     */
```

```
Fox(boolean randomAge, Field field, Location location) {
    // body of constructor omitted
}

/**
 * This is what the fox does most of the time: it hunts for rabbits.
 * In the process, it might breed, die of hunger, or die of old age.
 *
 * @param newFoxes
 *            A list to return newly born foxes.
 */
void hunt(List<Fox> newFoxes) {
    incrementAge();
    incrementHunger();
    if (alive) {
        giveBirth(newFoxes);
        // 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 = field.freeAdjacentLocation(location);
        }
        // See if it was possible to move.
        if (newLocation != null) {
            setLocation(newLocation);
        } else {
            // Overcrowding.
            setDead();
        }
    }
}

/**
 * Look for rabbits adjacent to the current location.
 * Only the first live rabbit is eaten.
 *
 * @return Where food was found, or null if it wasn't.
 */
private Location findFood() {
    List<Location> adjacent = field.adjacentLocations(location);
    Iterator<Location> it = adjacent.iterator();
    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;
                return where;
            }
        }
    }
    return null;
```

```
    }

    Other methods omitted.
}
```

Foxes **hunt** and **findFood**. If a rabbit is on an adjacent location, the fox eats it.

Exercises

14. What simplifications are made by the **Fox** class compared to real foxes? Do you feel that these could have a significant impact on the accuracy of the simulation?
15. Does increasing the maximum age for foxes lead to significantly higher numbers of foxes throughout a simulation, or is the rabbit population more likely to be reduced to zero as a result?
16. Experiment with different combinations of settings (breeding age, maximum age, breeding probability, litter size, etc.) for foxes and rabbits. Do species always disappear completely in some configurations? Are there configurations that are stable?
17. Experiment with different sizes of field. Does the size of the field affect the likelihood of species surviving?
18. Compare the results of running a simulation with a single large field and two simulations where each field is half the area of the large field. This somewhat models splitting a field with a highway. Are the simulations different?
19. Try the previous exercise with different size fields. Does size and shape matter?
20. Currently a fox will eat at most one rabbit at each step. Modify the **findFood** method so that rabbits in all adjacent locations are eaten at a single step. Assess the impact of this change on the results of the simulation.
21. When a fox eats a large number of rabbits at a single step, there are several different possibilities as to how we can model its food level. If we add all the rabbits' food values, the fox will have a very high food level, making it unlikely to die of hunger for a very long time. Alternatively, we could impose a ceiling on the fox's **foodLevel**. This models the effect of a predator that kills prey regardless of whether or not it is hungry. Assess the impacts of implementing this choice on the resulting simulation.
22. Given the randomness in the simulation, could a stable simulation eventually collapse?

The `Simulator` class: setup

```
Import statements and class comment omitted.

class Simulator {
    // static variables omitted

    // Lists of animals in the field. Separate lists are kept for
    // ease of iteration.
    private final List<Rabbit> rabbits;
    private final List<Fox> foxes;
    // 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 final SimulatorView view;
```

```
    /**
     * Construct a simulation field with default size.
     */
    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.
     */
    Simulator(int depth, int width) {
        if (width <= 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;
        }

        rabbits = new ArrayList<Rabbit>();
        foxes = new ArrayList<Fox>();
        field = new Field(depth, width);

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

        // Setup a valid starting point.
        reset();
    }

    /**
     * Run the simulation from its current state for the given number
     * of steps.
     * Stop before the given number of steps if it ceases to be viable.
     *
     * @param numSteps
     *            The number of steps to run for.
     */
    public void simulate(int numSteps) {
        for (int step = 1; step <= numSteps && view.isViable(field);
                step++) {
            simulateOneStep();
        }
    }

    /**
```

```
     * Run the simulation from its current state for a single step.
     * Iterate over the whole field updating the state of each fox
     * and rabbit.
     */
    public void simulateOneStep() {
        // method body omitted
    }

    /**
     * Reset the simulation to a starting position.
     */
    void reset() {
        step = 0;
        rabbits.clear();
        foxes.clear();
        populate();

        // Show the starting state in the view.
        view.showStatus(step, field);
    }

    /**
     * Randomly populate the field with foxes and rabbits.
     */
    private void populate() {
        Random rand = Randomizer.getRandom();
        field.clear();
        for (int row = 0; row < field.getDepth(); row++) {
            for (int col = 0; col < field.getWidth(); col++) {
                if (rand.nextDouble() <= FOX_CREATION_PROBABILITY) {
                    Location location = new Location(row, col);
                    Fox fox = new Fox(true, field, location);
                    foxes.add(fox);
                } else if (rand.nextDouble()
                            <= RABBIT_CREATION_PROBABILITY) {
                    Location location = new Location(row, col);
                    Rabbit rabbit = new Rabbit(true, field, location);
                    rabbits.add(rabbit);
                }
                // else leave the location empty.
            }
        }
    }

    Other methods omitted.
}
```

The simulator

1. constructs the field, the animal lists, the graphical interface,
2. populates the field with animals via the **`populate`** method,
3. simulates steps of the evolution via the **`simulateOneStep`** method.

Exercises

23. Modify the **populate** method of **Simulator** to determine whether or not setting an initial random age for foxes and rabbits is catastrophic.
24. If an initial random age is set for rabbits but not foxes, the rabbit population will tend to grow large while the fox population remains very small. Once the foxes do become old enough to breed, does the simulation tend to behave again like the original version? What does this suggest about the relative sizes of the initial populations and their impact on the outcome of the simulation?

The Simulator class: a simulation step

The **simulateOneStep** method determines what animals do at each step.

```
private void simulateOneStep() {
    step++;

    // Provide space for newborn rabbits.
    List<Rabbit> newRabbits = new ArrayList<>();
    // Let all rabbits act.
    for (Iterator<Rabbit> it = rabbits.iterator(); it.hasNext();) {
        Rabbit rabbit = it.next();
        rabbit.run(newRabbits);
        if (!rabbit.isAlive()) {
            it.remove();
        }
    }

    // Provide space for newborn foxes.
    List<Fox> newFoxes = new ArrayList<>();
    // Let all foxes act.
    for (Iterator<Fox> it = foxes.iterator(); it.hasNext();) {
        Fox fox = it.next();
        fox.hunt(newFoxes);
        if (!fox.isAlive()) {
            it.remove();
        }
    }

    // Add the newly born foxes and rabbits to the main lists.
    rabbits.addAll(newRabbits);
    foxes.addAll(newFoxes);

    view.showStatus(step, field);
}
```

Exercises

25. Each animal is contained in two different lists – in the field and on the **rabbits** or **foxes** lists in the **Simulator** class. Does this present a potential consistency problem between the lists used in **Field** and in the methods **simulateOneStep**, **hunt**, and **run**?
26. Instead of storing separate lists of rabbits and foxes, would it be better to generate

them from the field contents at the beginning of each simulation step? Discuss.

27. Test whether any (dead or alive) animals in the field which aren't included in the **rabbits** or **foxes** lists, and vice-versa. Are there any dead animals still listed anywhere?

Taking steps to improve the simulation

The most obvious shortcoming of the actual implementation is that we do not take account of the similarities between **Fox**es and **Rabbit**s.

Abstract classes

We will be refactoring the code in order to improve it, but this will not change the external behaviour of the application for the user. The first thing to note is that the **Fox** and **Rabbit** classes could be sensibly derived from some common superclass, for instance **Animal**. This could avoid code duplication by moving the common bits into the superclass.

Exercises

28. Identify the similarities and differences between the **Fox** and **Rabbit** classes: fields, methods, and constructors. Distinguish between class variables (static fields) and instance variables.
29. Candidate methods to place in a superclass are those that are identical in all subclasses. Which methods are truly identical in the **Fox** and **Rabbit** classes?
30. In the current version of the simulation, the values of all similarly named class variables are different. If the two values of a particlar class variable (**BREEDING_AGE**, say) were identical, would it make any difference to your assessment of which methods are identical?

The Animal superclass

- Attributes to move up to the **Animal** class: **alive**, **location**, along with methods: **isAlive**, and **setLocation**. For the moment we won't change **age.**
- What should the access levels be for the attributes of **Animal** in order to be available to methods of **Fox** and **Rabbit**? Declaring **age**, for instance, to be **protected** would work, but this would violate the principle of *loose coupling*. It is better to provide **Animal** with the appropriate setter and getter methods for the instance variables.
- Both animals have the instance variable **alive**; a **Fox** invokes a **Rabbit**'s **setEaten** method to set this variable to **false**. This could be replaced a more general method in **Animal**, eg, **setDead**.

Exercises

31. What sort of regression-testing strategy could you establish before undertaking the process of refactoring on the simulation? Is this something you could conveniently automate?
32. The **Randomizer** class determines whether “random” elements of the simulation are repeatable or not. If the **Randomizer.useShared** attribute is **true**, then a single

Random object is shared between all simulation objects. **Randomizer.reset** re-initializes the shared **Random** object. Use these to verify that introducing the **Animal** superclass does not change the simulation.

Create the **Animal** superclass in your version of the project. Make the changes discussed above. Ensure that the simulation works as before by checking the results at each step.

33. How has inheritance improved the project so far?

Abstract methods

Inheriting from **Animal** has allowed us to move some common code out of its subclasses. We still have a problem in the **Simulator** class, as illustrated by the following code.

```
for (Iterator<Animal> it = animals.iterator(); it.hasNext(); ) {
    Animal animal = it.next();
    if (animal instanceof Rabbit) {
        Rabbit rabbit = (Rabbit)animal;
        rabbit.run(newAnimals);
    } else if(animal instanceof Fox) {
        Fox fox = (Fox)animal;
        fox.hunt(newAnimals);
    } else {
        System.out.println("found unknown animal");
    }
    // Remove dead animals from the simulation.
    if (!animal.isAlive()) {
        it.remove();
    }
}
```

We can get an **Animal** object off the list, but we still need to identify its type in order to apply the appropriate method, **run** or **hunt**. That's what the **instanceof** operator does: **obj instanceof MyClass** returns **true** if **obj** is an object of type **MyClass**, and **false** otherwise. However, having to use **instanceof** is generally a BadIdea™ and a sign that something could be done better. Let's simplify the **Simulator** class to get rid of the **instanceof** operators in the **simulateOneStep** method. The above loop with type checks can be replaced by

```
for (Iterator<Animal> it = animals.iterator(); it.hasNext(); ) {
    Animal animal = it.next();
    animal.act(newAnimals);
    // Remove dead animals from the simulation.
    if (!animal.isAlive()) {
        it.remove();
    }
}
```

Note:

- The object returned from the animal list is of type **Animal**.
- The specific methods **hunt** and **run** for **Fox** and **Rabbit** respectively have been

replaced with a general method, **act**. More on this below.

- Through dynamic typing, **animal.act** will invoke **Fox**'s **act** method when **animal** is a **Fox**, and **Rabbit**'s **act** when **animal** is a **Rabbit**.
- To compile, **Animal** must have an **act** method with the appropriate signature.

The problem now is that we don't know how to define an **act** method for **Animal**, since every specific subtype of **Animal** behaves differently, and we'll never have an instance of an **Animal**. Since we don't know how to code this method, we'll declare it to be an abstract method, ie, a method with no code!

```
abstract void act(List newAnimals);
```

An abstract method:

- has the keyword **abstract**;
- has no code body. Hence, an abstract method can't be executed.

Abstract classes

If a class contains an abstract method, then the class is an abstract class, eg,

```
abstract class Animal {
    // fields omitted

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

    Other methods omitted.
}
```

- No instances of abstract classes can be created.
- Subclasses of abstract classes must override all abstract methods in order to be concrete (non-abstract). If a class doesn't override abstract methods, then the class is itself abstract.

Abstract classes determine behaviour for their subclasses by imposing methods that the subclasses *must* implement. For instance, any object declared to be of type **Animal** must have some implementation of the **act** method.

Exercises

34. Although the body of the loop in the **oneSimulationStep** method above no longer deals with **Fox** and **Rabbit** types, it still deals with the **Animal** type. Why is it not possible for it to treat each object in the collection simply using the **Object** type?

35. Is it necessary for a class with one or more abstract methods to be defined as abstract? Experiment.
36. Is it possible for a class that has no abstract methods to be described as abstract? Experiment.
37. Could it ever make sense to define a class as abstract if it has no abstract methods? Discuss.
38. Which classes in the **`java.util`** package are abstract? Some of them have **`abstract`** in the class name, but is there any other way to tell from the documentation? Which concrete classes extend them?
39. Can you tell from the API documentation for an abstract class which (if any) of its methods are abstract? Do you *need* to know which methods are abstract?
40. Review the overriding rules for methods and fields discussed in More about inheritance. Why are they particularly significant in our attempts to introduce inheritance into this application?
41. The changes made in this section have removed the dependencies (couplings) of the **`simulateOneStep`** method to the **`Fox`** and **`Rabbit`** classes. The **`Simulator`** class, however, is still coupled to **`Fox`** and **`Rabbit`** because these classes are references in the **`populate`** method. There is no way to avoid this; when we create animal instances, we have to specify exactly what kinds of animals to create.

    This could be improved by splitting the **`Simulator`** into two classes: one class, **`Simulator`**, which runs the simulation and is completely decoupled from the concrete animal classes, and another class, **`PopulationGenerator`** (created and called by the simulator), which creates the populations. Only this class need be coupled to the concrete animal classes, making it easier for a maintenance programmer to find places where change is necessary when the application is extended. Try implementing this refactoring step. The **`PopulationGenerator`** class should also define the colours for each type of animal.

The project *foxesandrabbits.v2* implements the above changes to improve the simulation. Note that because all animals are handled by a single list, the results of the two versions will be different.

We will look at the project *foxesandrabbits.graph* which presents another view of the simulation.

Exercises

42. Run the *foxesandrabbits.graph* project. What do the curves represent and what is the relationship between them?
43. Repeat the simulation with different size fields. Does the graph view help to understand the simulations?

More abstract methods

We couldn't move the **`canBreed`** method into **`Animal`** because it uses the **`BREEDING_AGE`** constant, and this is different for **`Fox`** and **`Rabbit`**. Fields (instance variables or static variables) are not like methods because they are never overridden by subclass types. Hence the **`BREEDING_AGE`** for both subclasses would be that of **`Animal`**, and that's not what we want. However, the following *can* go into **`Animal`**

```
/**
 * An animal can breed if it has reached the breeding age.
 * @return true If the animal can breed
 */
boolean canBreed() {
    return age >= getBreedingAge();
}

abstract int getBreedingAge();
```

provided the subclasses override **getBreedingAge**, eg,

```
/**
 * @return The age at which a rabbit can breed
 */
int getBreedingAge() {
    return BREEDING_AGE;
}
```

Exercises

44. Using your latest version of the project, move the **canBreed** method from **Fox** and **Rabbit** to **Animal** and rewrite it.
45. Move the **age** attribute from **Fox** and **Rabbit** into **Animal**.Initialize it to zero in the constructor. Supply getter and setter methods, and use these methods in **Fox** and **Rabbit** to access the attribute. Does the application still compile and does it work as before?
46. Move the **canBreed** method from **Fox** and **Rabbit** into **Animal** **and rewrite it as above. Supply appropriate versions of** **getBreedingAge** **in** **Fox** **and** **Rabbit** **to return each category's breeding age.**
47. Move the **incrementAge** method from **Fox** and **Rabbit** to **Animal** by providing an abstract **getMaxAge** method in **Animal** and a concrete version in **Fox** and **Rabbit**.
48. Can the **breed** method be moved to **Animal**? If so, make this change.
49. In light of the changes you have made to these three classes, reconsider the visibility of each method and make any changes you feel are appropriate.
50. Was it possible to make these changes without having any impact on the other classes in the project? If so, what does this suggest about the degrees of encapsulation and coupling that were present in the original version?
51. Define a completely new type of animal for the simulation as a subtype of **Animal**. You will need to decide what sort of impact its existence will have on the existing animal types. For instance, your animal might compete with foxes as a predator on the rabbit population, or your animal might prey on foxes but not on rabbits. You will probably find that you need to experiment quite a lot with the configuration settings you use for it. You will need to modify the **populate** method to have some of your animals created at the start of a simulation.
52. Challenge exercise – the code for the **giveBirth** methods in **Fox** and **Rabbit** are pretty similar, they just create different types of animals. Can this method be moved into **Animal**?

You should also define a new colour for your new animal class. You can find a list of predefined colour names on the API page documenting the **Color** class in the **java.awt**

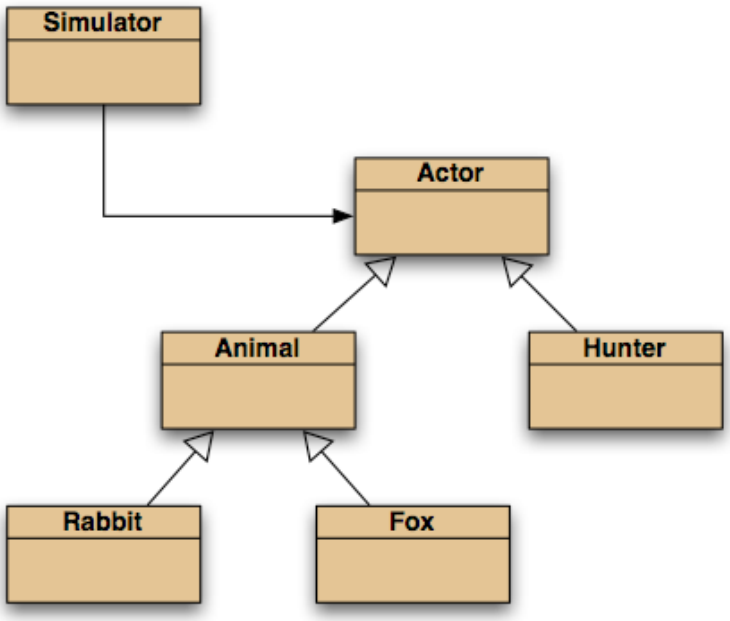
package.

Multiple inheritance

An `Actor` class

We can image adding new entities into our simulation which might not be **`Animal`**s, for example **`Hunter`**, or maybe not even something animate, like the weather. Whatever we add will somehow act in the simulation, so we could generalize it as an **`Actor`** class.

This might look something like:




```
// all comments omitted

abstract class Actor {
    abstract void act(List<Actor> newActors);
    abstract boolean isActive();
}
```

Exercise

53. Introduce the **`Actor`** class into your simulation. Rewrite the **`simulateOneStep`** method in **`Simulator`** to use **`Actor`** instead of **`Animal`**. You can do this even if you have not introduced any new participant types. Does the **`Simulator`** class compile? Or is there something else that is needed in the **`Actor`** class?

Flexibility through abstraction

We've pushed abstraction quite far, now we'll go even further by separating visualization from acting. This is a common step in building serious applications.

Selective drawing

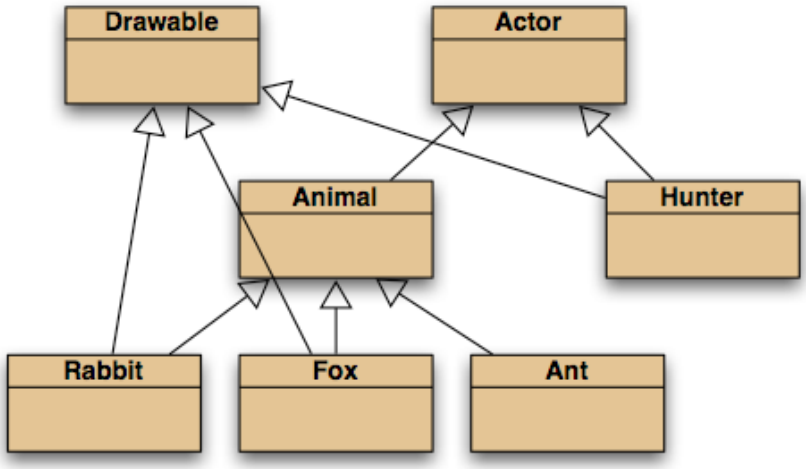
We can image that things to be visualized know how to draw themselves, so that at every iteration an object would act and then draw itself:

```
// let all actors act
for (Actor actor : actors) {
    actor.act(...);
}

// draw all drawables
for (Drawable item : drawables) {
    item.draw(...);
}
```

Drawable actors: multiple inheritance

In this case, drawable actors would inherit from both **`Actor`** and **`Drawable:`**



Unlike some other object-oriented languages, Java does not permit multiple inheritance of code. However, what we need for the above extension to **`Simulator`** is not multiple inheritance of code but rather *multiple inheritance of behaviour*, and this is best accomplished through the mechanism of *interfaces*.

Interfaces

The informal notion of interface as we have used it so far is formalized in Java into the **`interface`** structure. These are sort of abstract classes where all the methods are abstract.

An `Actor` interface

```
/**
```

```
 * The interface to be implemented by any class wishing
 * to participate in the simulation.
 */
interface Actor {
    /**
     * Perform the actor's regular behavior.
     * @param newActors A list for receiving newly created actors.
     */
    void act(List<Actor> newActors);

    /**
     * Is the actor still active?
     * @return true If still active, false otherwise.
     */
    boolean isActive();
}
```

- The declaration says **interface** rather than **class**.
- All methods are abstract so the keyword **abstract** can be omitted.
- All methods are public so the keyword **public** can be omitted.
- No constructor.
- Only constant fields are permitted. **public**, **static** and **final** keywords may be omitted.

A class *implements* an interface, eg,

```
class Fox extends Animal implements Drawable { ... }
```

Exercises

54. Redefine the abstract class **Actor** in your project as an interface. Does the simulation still compile? Does it run?
55. Are the fields in the following interface static fields or instance fields?

```
interface Quiz {
    int CORRECT = 1;
    int INCORRECT = 0;
}
```

    What visibility do they have?

56. What are the errors in the following interface?

```
interface Monitor {
    private static final int THRESHOLD = 50;

    private Monitor(int initial);

    int getThreshold() {
        return THRESHOLD;
    }
}
```

Default methods in interfaces

A method marked as `default` in an interface will have a method body, which is inherited by all implementing classes. The addition of default methods to interfaces in Java 8 muddied the waters in the distinction between abstract classes and interfaces, since it is no longer true that interfaces never contain method bodies. However, it is important to exercise cau- tion when considering defining a default method in an interface. It should be borne in mind that default methods were added to the language primarily to support the addition of new methods to interfaces in the API that existed before Java 8. Default methods made it possible to change existing interfaces without breaking the many classes that already implemented the older versions of those interfaces.

From the other limitations of interfaces—no constructors and no instance fields—it should be clear that the functionality possible in a default method is strictly limited, since there is no state that can be examined or manipulated directly by them. In general, therefore, when writing our own interfaces we will tend to limit ourselves to purely abstract methods. Furthermore, when discussing features of interfaces in this chapter we will often ignore the case of non-abstract methods for the sake of simplicity.

Multiple inheritance of interfaces

A class can implement multiple interfaces, eg,

```
class Hunter implements Actor, Drawable { ... }
```

The class **`Hunter`** must supply the methods declared in both interfaces, or it must be declared abstract.

Exercise

57. Challenge exercise - Add a non-animal actor to the simulation. For instance, you could introduce a **`Hunter`** class with the following properties. Hunters have no maximum age and neither feed nor breed. At each step of the simulation, a hunter moves to a random location anywhere in the field and fires a fixed number of shots into random target locations around the field. Any animal in one of the target locations is killed.

    Place just a small number of hunters in the field at the start of the simulation. Do the hunters remain in the simulation throughout, or do they sometimes disappear? If they do disappear, why might that be, and does that represent realistic behaviour?

    What other classes required changing as a result of introducing hunters? Is there a need to introduce further decoupling to the classes?

Interfaces as types

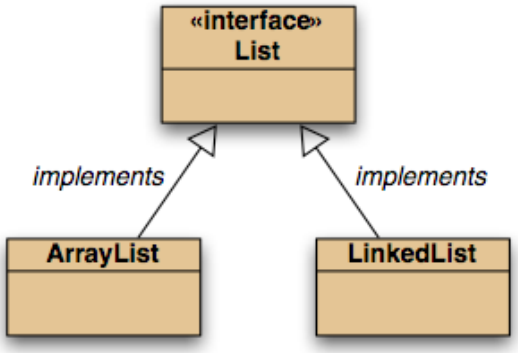
In a previous section, *Improving structure with inheritance*, we saw the advantages of inheritance:

- avoiding code duplication since the subclasses inherit code from their superclass;
- the superclass becomes a subtype of the superclass thus allowing polymorphism. This can simplify application structure.

Since the first point, inheritance of implementation, doesn't apply to interfaces, only the second point applies: interfaces serve as supertypes for instances of other classes.

Interfaces as specifications

Interfaces completely separate behaviour from implementation. They can allow the development of a specification to which an implementation can then be *plugged in*. For instance, from the collections framework we have



In our application, we would write

```
private List<SomeType> myList = new LinkedList<>();
```

and then treat the variable **`myList`** as of type **`List`**. If we later decide that we'd really prefer using an **`ArrayList`** (which is faster for some operations, but slower for others), then we'd have only one line of code to change

```
private List<SomeType> myList = new ArrayList<>();
```

(there also exist techniques for avoiding changing any application code whatsoever). Effectively, we'd *plug in* the **`ArrayList`** implementation of **`List`** in place of the **`LinkedList`** implementation.

Exercises

58. Which methods do **`ArrayList`** and **`LinkedList`** have that are not defined in the **`List`** interface? Why do you think these methods are not included in **`List`**?
59. Write a class that can make comparisons between the efficiency of the common methods from the **`List`** interface in the **`ArrayList`** and **`LinkedList`** classes such as **`add`**, **`get`**, and **`remove`**. Use the polymorphic-variable technique described above to write the class so that it only knows it is performing its tests on objects of the interface type **`List`** rather than on the concrete types **`ArrayList`** and **`LinkedList`**. Use large lists of objects for the tests, to make the results significant. You can use the **`currentTimeMillis`** method of the **`System`** class for getting hold of the start and finish time of your test methods.
60. Read the API description for the **`sort`** methods of the **`Collections`** class in the **`java.util`** package. Which interfaces are mentioned in the descriptions?
61. Challenge exercise - Investigate the **`Comparable`** interface. This is a parametrized interface. Define a simple class that implements **`Comparable`**. Create a collection containing objects of this class and sort the collection. *Hint:* The **`LogEntry`** class of the *webloganalyzer* project implements this interface.

Library support through abstract classes and interfaces

Functional interfaces and lambdas (advanced)

Java 8 introduced a special classification for interfaces that contain just a single abstract method (regardless of the number of default and/or static methods they contain). Such an interface is called a *functional interface*. The annotation **`@FunctionalInterface`** may be included with the declaration to allow the compiler to check that the interface conforms to the rules for functional interfaces.

There is a special relationship between functional interfaces and lambda expressions. Anywhere that an object of a functional interface type is required, a lambda expression may be used instead. In the following chapter, we shall see extensive use of this feature when implementing graphical user interfaces.

This link between lambdas and functional interfaces is important. In particular, it gives us a convenient means to associate a lambda expression with a type, for example to declare a variable that can hold a lambda. The **`java.util.function`** package defines a large number of interfaces that provide convenience names for the most commonly occurring types of lambda expression. In general, the interface names indicate the return type and the parameter types of their single method, and hence a lambda of that type. For example:

- **`Consumer`** interfaces relate to lambdas with a void return type. For instance, **`DoubleConsumer`** takes a single **`double`** parameter and returns no result.
- **`BinaryOperator`** interfaces take two parameters and return a result of the same type. For instance, **`IntBinaryOperator`** takes two **`int`** parameters and returns an **`int`** result.
- **`Supplier`** interfaces return a result of the indicated type. For instance, **`LongSupplier`**.
- **`Predicate`** interfaces return a **`boolean`** result. For instance, **`IntPredicate`**.

All of these interfaces extend the **`Function`** interface, whose single abstract method is called **`apply`**. One potentially useful functional interface type, defined in the j**`ava.lang`** **package**, is **`Runnable`**. This fills a gap in the list of **`Consumer`** interfaces in that it takes no parameters and has a **`void`** return type. However, its single abstract method is called **`run`**.

Functional interface types allow lambdas to be assigned to variables or passed as actual parameters. For example, suppose we have pairs of *name* and *alias* strings that we wish to format in a particular way, such as **`"Michelangelo Merisi (AKA Caravaggio)"`**. A lambda taking two **`String`** parameters and returning a **`String`** result is compatible with the **`BinaryOperator`** interface, and we might give a type and name to a lambda as follows:

```
BinaryOperator<String> aka = (name, alias) -> return name + " (AKA " +
alias + ")";
```

This lambda would be used by calling its **`apply`** method with appropriate parameters, for instance:

```
System.out.println(aka.apply("Michelangelo Merisi", "Caravaggio"));
```

A further example of interfaces

In the previous section we discussed how interfaces can be used to separate the specification of a component from its implementation, so that different implementations can be "plugged in", thus making it easy to replace components of a system.

When we separate the visualization of the simulation from the logic (the actors and actions), we could easily plug in different visualizations

- as a plot of population of species as a function of time (steps);
- as a purely textual output.

Exercises

62. Review the source code of the **`Simulator`** class. Find all occurrences of the view classes and interfaces, and trace all variables declared using any of these types. Explain exactly how the views are used in the **`Simulator`** class.
63. Implement a new class **`TextView`** that implements **`SimulatorView`**. **`TextView`** provides a textual view of the simulation. After every simulation step, it prints out one line in the form

    ```
    Foxes: 121    Rabbits: 266
    ```

    Use **`TextView`** instead of **`AnimatedView`** for some tests. (Do not delete the **`AnimatedView`** classes. We want to have the ability to change between both views!)
64. Can you manage to have both views active at the same time?

The `Class` class

We have seen previously the class Object (Java's cosmic superclass). Well, there's also a class called...Class. Each type in Java has an associated Class object. The Object method getClass returns an object type's Class, as does, for example Fox.class. The Class class can be useful to differentiate between different classes, for instance as keys for a map in SimulatorView:

```
private Map<Class, Color> colors;
...
view.setColor(Rabbit.class, Color.ORANGE);
view.setColot(Fox.class, Color.BLUE);
```

Abstract class or interface?

When subclasses must inherit code, you need to use an abstract class; otherwise interfaces are preferred.

Event-driven simulations

The style of simulation we have used in this chapter has the characteristic of time passing in discrete, equal-length steps. At each time step, each actor in the simulation was asked to act - i.e., take the actions appropriate to its current state. This style of simulation is sometimes called time-based, or synchronous, simulation. In this particular simulation, most of the actors will have had something to do at each time step: move, breed, and eat. In many simulation scenarios, however, actors spend large numbers of time steps doing nothing - typically, waiting for something to happen that requires some action on their part. Consider

the case of a newly born rabbit in our simulation, for instance. It is repeatedly asked whether it is going to breed, even though it takes several time steps before this is possible. Isn't there a way to avoid asking this unnecessary question until it is actually ready?

There is also the question of the most appropriate size of the time step. We deliberately left vague the issue of how much real time a time step represents, and the various actions actually require significantly different amounts of time (eating and movement should occur much more frequently than giving birth, for instance). Is there a way to decide on a time-step size that is not so small that most of the time nothing will be happening or too long that different types of actions are not distinguished clearly enough between time steps?

An alternative approach is to use an event-based, or asynchronous, simulation style. In this style, the simulation is driven by maintaining a schedule of future events. The most obvious difference between the two styles is that, in an event-based simulation, time passes in uneven amounts. For instance, one event might occur at time t and the next two events occur at times t+2 and time t+8, while the following three events might all occur at time t+9.

For a fox-and-rabbits simulation, the sort of events we are talking about would be birth, movement, hunting, and death from natural causes. What typically happens is that, as each event occurs, a fresh event is scheduled for some point in the future. For instance, when a birth event occurs, the event marking that animal's death from old age will be scheduled. All future events are stored in an ordered queue, where the next event to take place is held at the head of the queue. It is important to appreciate that newly scheduled events will not always be placed at the end of the current queue; they will often have to be inserted somewhere before the end, in order to keep the queue in time order. In addition, some future events will be rendered obsolete by events that occur before them—an obvious example is that the natural-death event for a rabbit will not take place if the rabbit is eaten beforehand!

Event-driven simulations lend themselves particularly well to the techniques we have described in this chapter. For instance, the concept of an event is likely to be implemented as an **`Event`** abstract class containing concrete details of when the event will occur, but only abstract details of what the event involves. Concrete subclasses of **`Event`** will then supply the specific details for the different event types. Typically, the main simulation loop will not need to be concerned with the concrete event types, but will be able to use polymorphic method calls when an event occurs.

Event-based simulations are often more efficient and are preferable where large systems and large amounts of data are involved, while synchronous simulations are better for producing time-based visualizations (such as animations of the actors) because time flows more evenly.

Exercises

65. Find out some more about how event-driven simulations differ from time-based simulations.
66. Look at the **`java.util`** package to see if there are any classes that might be well suited to storing an event queue in an event-based simulation.
67. Challenge exercise: Rewrite the foxes-and-rabbits simulation in the event-based style.

Summary of inheritance

Two main purposes of inheritance

- code inheritance for code reuse
    - by extending concrete classes;

- type inheritance for polymorphism
    - by extending abstract classes and overriding abstract methods;
    - by implementing interfaces.

We can override methods inherited from concrete classes, and we can add new methods in subclasses. In order to instantiate classes which inherit abstract methods, all such methods must be implemented.

Summary

Concept summary

abstract method
: An abstract method definition consists of a method signature without a method body. It is marked with the key word **`abstract`**.

abstract class
: An abstract class is a class that is not intended for creating instances. Its purpose is to serve as a superclass for other classes. Abstract classes may contain abstract methods.

abstract subclasses
: For a subclass of an abstract class to become concrete, it must provide implementations for all inherited methods. Otherwise it will itself be abstract.

superclass method calls
: Calls to non-private instance methods from within a superclass are always evaluated in the wider context of the object's dynamic type.

multiple inheritance
: A situation in which a class inherits from more than one superclass is called multiple inheritance).

interface
: A Java interface is a specification of a type (in the form of a type name and a set of methods) that does not define any implementation for any of the methods.

Additional exercises

Exercises

68. Can an abstract class have concrete (non-abstract) methods? Can a concrete class have abstract methods? Can you have an abstract class without abstract methods? Justify your answers.
69. Look at the code below. You have five types (classes or interfaces) (**`U`**, **`G`**, **`B`**, **`Z`**, and **`X`**) and a variable of each of these types.

```
U u;
G g;
B b;
```

```
Z z;
X x;
```

The following assignments are all legal.

```
u = z;
x = b;
g = u;
x = u;
```

The following assignments are all illegal (they cause compiler errors).

```
u = b;
x = g;
b = u;
z = u;
g = x;
```

What can you say about the types and their relationships? (What relationships are they to each other?)

70. Assume you want to model people in a university to implement a course management system. There are different people involved: staff members, students, teaching staff, support staff, tutors, technical support staff, and student technicians. Tutors and student technicians are interesting: tutors are students who have been hired to do some teaching, and student technicians are students who have been hired to help with technical support. Draw a type hierarchy (classes and interfaces) to represent this situation. Indicate which types are concrete classes, abstract classes, and interfaces.

71. Sometimes class/interface pairs exist in the Java standard library that define exactly the same methods. Often, the interface name ends with *Listener* and the class name with *Adapter*. An example is **`PrintJobListener`** and **`PrintJobAdapter`**. The interface defines some method signatures, and the adapter class defines the same methods, each with an empty method body. What might the reason be for having them both?

72. The collection library has a class named **`TreeSet`**, which is an example of a sorted set. Elements in this set are kept in order. Read the description of this class carefully, and then write a class **`Person`** that can be inserted into a **`TreeSet`**, which will then sort the **`Person`** objects by age.

73. Use the API documentation for the **`AbstractList`** class to write a concrete class that maintains an unmodifiable list.