Object-Oriented Programming with Java, part II ››

Authors: Arto Vihavainen, Matti Luukkainen
Translators: Simone Romeo, Kenny Heinonen

Exercise 27: The Finnish Ringing Centre

In the sixth week of Introduction to Programming, we created an observation database for bird watching. Now we continue in the same vein, and this time, we create a program for the ringing centre to track of the places where the rung birds were observed.

ATTENTION: You may run into a strange error message in this exercise, such as `NoSuchMethodError: Bird.equals(BBird;)Z`; if this happens, **clean and build**, i.e. press the brush and hammer icon in NetBeans.

EXERCISE 27.1: BIRD EQUALS AND TOSTRING

The Finnish Ringing Centre stores the information about the birds who were watched in a specific year in `Bird` objects:

```
public class Bird {

    private String name;
    private String latinName;
    private int ringingYear;
```

```
    public Bird(String name, String latinName, int ringingYear) {
        this.name = name;
        this.latinName = latinName;
        this.ringingYear = ringingYear;
    }

    @Override
    public String toString() {
        return this.latinName + "(" + this.ringingYear + ")";
    }
}
```

The idea is implementing the funcionality for the ringing centre to track of the places where rung birds were observed and how many times they were. However, observation places and times are not stored in Bird objects, but in a separate HashMap, whose keys are Bird objects. As we remember from Week 2, in such cases we have to implement the methods `equals(Object other)` and `hashCode()` in the class `Bird`.

Some birds have more than one English name (for instance, the Rose Starling is also known as Rose-Coloured Starling or Rose-Coloured Pastor); however, the Latin name is always unique. Create the methods `equals` and `hashCode`- in the class `Bird`; two Bird objects have to be understood as the same bird if their Latin name and observation year are the same.

Example:

```
Bird bird1 = new Bird("Rose Starling", "Sturnus roseus", 2012);
Bird bird2 = new Bird("Rose-Coloured Starling", "Sturnus roseus", 2012);
Bird bird3 = new Bird("Hooded Crow", "Corvus corone cornix", 2012);
Bird bird4 = new Bird("Rose-Coloured Pastor", "Sturnus roseus", 2000);

System.out.println( bird1.equals(bird2));   // same Latin name and same obse
System.out.println( bird1.equals(bird3));   // different Latin name: they are
System.out.println( bird1.equals(bird4));   // different observation year: no
System.out.println( bird1.hashCode()==bird2.hashCode() );
```

Prints:

```
true
false
false
true
```

EXERCISE 27.2: RINGING CENTRE

The Ringing Centre has two methods: `public void observe(Bird bird, String place)`, which records the observation and its place among the bird information; and `public void observations(Bird bird)`, which prints all the observations of the parameter bird following the example below. The observation printing order is not important, as far as the tests are concerned.

The Ringing Centre stores the observation places in a `Map<Bird, List<String>>` object variable. If you need, you can use the exercise from Section 16 as example.

An example of how the Ringing Centre works:

```
RingingCentre kumpulaCentre = new RingingCentre();

kumpulaCentre.observe( new Bird("Rose Starling", "Sturnus roseus", 2012), "Arabia" )
kumpulaCentre.observe( new Bird("Rose-Coloured Starling", "Sturnus roseus", 2012), "
kumpulaCentre.observe( new Bird("European Herring Gull", "Larus argentatus", 2008),
kumpulaCentre.observe( new Bird("Rose Starling", "Sturnus roseus", 2008), "Mannerhei

kumpulaCentre.observations( new Bird("Rose-Coloured Starling", "Sturnus roseus", 201
System.out.println("--");
kumpulaCentre.observations( new Bird("European Herring Gull", "Larus argentatus", 20
System.out.println("--");
kumpulaCentre.observations( new Bird("European Herring Gull", "Larus argentatus", 19
```

Prints:

```
Sturnus roseus (2012) observations: 2
Arabia
Vallila
--
Larus argentatus (2008) observations: 1
Kumpulanmäki
--
Larus argentatus (1980) observations: 0
```

49. OBJECT POLYMORPHISM

Precedently, we have run into situations where variables had various different types, in addition to their own. For instance, in the section 45, we noticed that *all* objects are `Object`-type. If an object is a particular type, we can also represent it as `Object`-type. For instance, `String` is also `Object`-type, and all `String` variables can be defined using `Object`.

```
String string = "string";
Object string = "another string";
```

It is possible to assign a String to an `Object`-type reference.

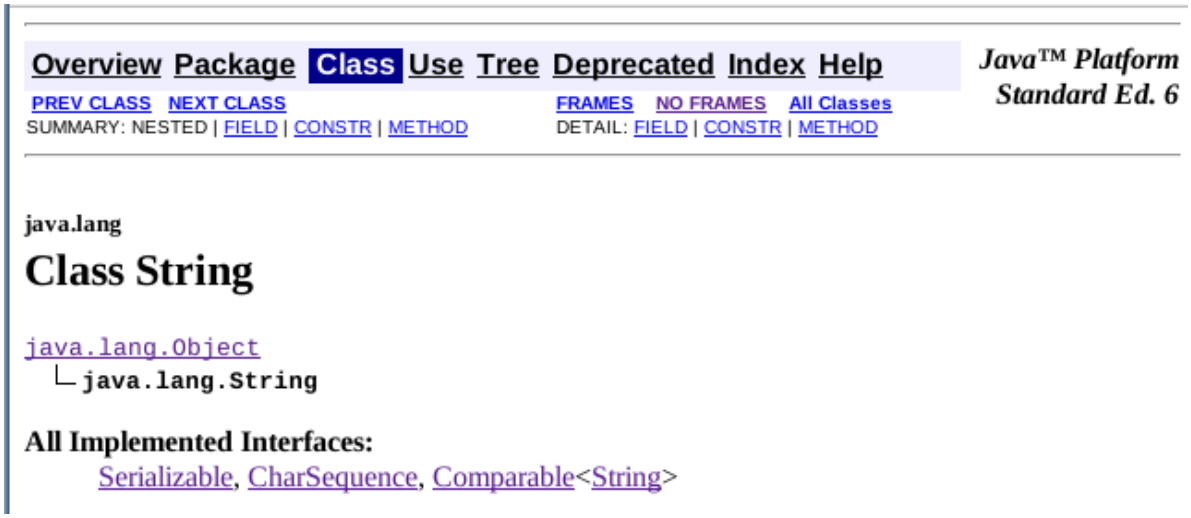
```
String string = "characterString";
Object string = string;
```

The opposite way does not work. Because `Object`-type variables are not Strings, an Object variable cannot be assigned to a String variable.

```
Object string = "another string";
String string = string; // DOESN'T WORK!
```

What is the real problem?

Variables have got their own type, and in addition to it they also have got the type of their parent classes and interfaces. The class `String` derives from the `Object` class, and therefore `String` objects are also `Object`-type. The class `Object` does not derive from the class `String`, and therefore `Object` variables are not automatically `String`-type. Let's dig deeper into the `String` class API documentation, expecially the upper part of the HTML page.



The String class API documentation starts with the common heading; this is followed by the class package (`java.lang`). After the package you find the class name (`Class String`), and this is followed by the *inheritance hierarchy*.

```
java.lang.Object
  └java.lang.String
```

The inheritance hierarchy lists the classes from which a class derives. The inherited classes are listed in hierarchical order, where the class we are analizing is the last one. As far as our String class inheritance hierarchy is concerned, we notice that the `String` class derives from the class `Object`. *In Java, each class can derive from one class, tops*; however, they can inherit features of more than one, undirectly.

You can think inheritance hierarchy as if it was a list of types, which the object has to implement.

The fact that all objects are `Object`-type helps programming. If we only need the features defined in the `Object` class in our method, we can use `Object` as method parameter. Because all objects are also Object-type, a method can be given *whatever* object as parameter. Let's create the method `printManyTimes`, which receives an `Object` variable as parameter, and the number of times this must be printed.

```
public class Printer {
    ...
    public void printManyTimes(Object object, int times) {
        for (int i = 0; i < times; i++) {
            System.out.println(object.toString());
        }
    }
    ...
}
```

You can give whaterver object parameter to the method `printManyTimes`. Within the method `printManyTimes`, the object has only the method which are defined in the `Object` class at its disposal, because the method is *presented* as `Object`-type inside the mehod.

```
    Printer printer = new Printer();

    String string = " o ";
    List<String> words = new ArrayList<String>();
    words.add("polymorphism");
    words.add("inheritance");
    words.add("encapsulation");
    words.add("abstraction");

    printer.printManyTimes(string, 2);
    printer.printManyTimes(words, 3);
```

```
 o
 o
[polymorphism, inheritance, encapsulation, abstraction]
[polymorphism, inheritance, encapsulation, abstraction]
[polymorphism, inheritance, encapsulation, abstraction]
```

Let's continue with our `String` class API inspection. In the description, the inheritance hierarchy is followed by a list of the interfaces which the class implements.

```
All Implemented Interfaces:
  Serializable, CharSequence, Comparable<String>
```

The `String` class implements the interfaces `Serializable`, `CharSequence`, and `Comparable<String>`. An interface is a type, too. According to the description of the String API, we should be able to set the following interfaces as the type of a String object.

```
    Serializable serializableString = "string";
    CharSequence charSequenceString = "string";
    Comparable<String> comparableString = "string";
```

Because we can define the parameter type of a method, we can define methods which would accept an object which implements *a specific interface*. When we define an interface as method

parameter, the parameter can be whatever object which implements such interface, the method does not care about the object actual type.

Let's implement our `Printer` class, and create a method to print the characters of the objects which implement the interface `CharSequence`. The `CharSequence` interface also provides methods such as `int length()`, which returns the String's length, and `char charAt(int index)`, which returns the character at a specific index.

```
public class Printer {
    ...
    public void printManyTimes(Object object, int times) {
        for (int i = 0; i < times; i++) {
            System.out.println(object.toString());
        }
    }

    public void printCharacters(CharSequence charSequence) {
        for (int i = 0; i < charSequence.length(); i++) {
            System.out.println(charSequence.charAt(i);
        }
    }
    ...
}
```

Whatever object which implements the `CharSequence` interface can be assigned to the method `printCharacters`. For instance, you can give a `String` or a `StringBuilder` which is usually more efficient when it comes to string building. The method `printCharacters` prints each character of the object in its own line.

```
    Printer printer = new Printer();

    String string = "works";

    printer.printCharacters(string);
```

```
w
o
r
k
s
```

Exercise 28: Groups

In this exercise, we make organisms and groups of organisms which move around each other. The position of the organisms is reported by using a *bidimensional coordinate system*. Each position is defined by two numbers, the `x` and `y` coordinates. The `x` coordinate tells us how far from the "point zero" the position is horizontally, whereas the

`y` coordinate tells u how far the position is vertically. If you have got doubts of what a coordinate system is, you can read more information in Wikipedia, for instance.

Together with the exercise, you find the interface `Movable`, which represents things that can be moved from one place to another. The interface contains the method `void move(int dx, int dy)`. The parameter `dx` tells us how much the object moves on the x axis and `dy` tells us about the movement on the y axis.

Implement the classes `Organism` and `Group`, which are both movable. Implement all the functionality inside the package `movable`.

EXERCISE 28.1: IMPLEMENTING ORGANISM

Create the class `Organism` in the package `movable`; let Organism implement the interface `Movable`. Organisms have to know their own position (x and y coordinates). The API of `Organism` has to be the following:

- the constructor **public Organism(int x, int y)**
  ; it receives the x and y initial coordinates of the object
- **public String toString()**
  ; it creates and returns a string which represents the object. The form should be the following `"x: 3; y: 6"`. Note that the coordinates are separated by a semicolon (;)
- **public void move(int dx, int dy)**
  ; it moves the object as much as it is specified by the arguments. The variable `dx` contains the x coordinate of the movement, whereas `dy` contains the y coordinate of the movement. For instance, if the value of the variable `dx` is 5, the object variable `x` has to be increased by five

Try our the functionality of `Organism` using the following code.

```
Organism organism = new Organism(20, 30);
System.out.println(organism);
organism.move(-10, 5);
System.out.println(organism);
organism.move(50, 20);
System.out.println(organism);
```

```
x: 20; y: 30
x: 10; y: 35
x: 60; y: 55
```

EXERCISE 28.2: IMPLEMENTING GROUP

Create the class `Group` in the package `movable`; Group implements the interface `Movable`. The Group is made of various different objects which implement the interface `Movable`, and they have to be stored into a list construction, for instance.

The class `Group` should have the following API.

- **public String toString()**
  ; it returns a string which describes the position of the group organisms, each organism is printed in its own line.
- **public void addToGroup(Movable movable)**
  ; it adds a new objects which implements the interface `Movable` to the group.
- **public void move(int dx, int dy)**
  ; it moves a group as much as it is defined by the arguments. Note that you will have to move each group organism.

Try out your program functionality with the following code.

```
Group group = new Group();
group.addToGroup(new Organism(73, 56));
group.addToGroup(new Organism(57, 66));
group.addToGroup(new Organism(46, 52));
group.addToGroup(new Organism(19, 107));
System.out.println(group);
```

```
x: 73; y: 56
x: 57; y: 66
x: 46; y: 52
x: 19; y: 107
```

50. INHERITANCE OF CLASS FEATURES

Classes are for the programmer a way to clarify problematic concepts. With each class we create, we add new functionality to the programming language. The functionality is needed to solve the problems we meet, and the solutions are born from the interaction among the objects we create. In object programming, an object is an independent unity which can change through its methods. Objects are used together with each other; each object has its own area of responsibility. For instance, our user interface classes have been making use of `Scanner` objects, so far.

Each Java's class descends from the class `Object`, which means that each class we create has all methods which are defined in `Object`. If we want to change the functionality of the methods defined in `Object`, we have to `Override` them and define a new functionality in the created class.

In addition to be possible to inherit the `Object` class, it is also possible to inherit other classes. If we check Java's `ArrayList` class API we notice that `ArrayList` inherits the class `AbstractList`. The class `AbstractList` inherits the class `AbstractCollection`, which descended from the class `Object`.

```
java.lang.Object
  └java.util.AbstractCollection<E>
```

```
    └java.util.AbstractList<E>
        └java.util.ArrayList<E>
```

Each class can inherit one class, directly, Indirectly, a class can still inherit all the feauters its parent class. The class `ArrayList` inherits directly the class `AbstractList`, and indirectly the classes `AbstractCollection` and `Object`. In fact, the class `ArrayList` has the methods *and* interfaces of `AbstractList`, `AbstractCollection` and `Object` at its disposal.

The class features are inherited using the keyword `extends`. The class which inherits is called *subclass*; the class which is inherited is called *superclass*. Let's get aquainted with a carmaker system, which handles car components. The basic component of component handling is the class `Component` which defines the identification number, the producer, and the description.

```
public class Component {

    private String id;
    private String producer;
    private String description;

    public Component(String id, String producer, String description) {
        this.id = id;
        this.producer = producer;
        this.description = description;
    }

    public String getId() {
        return id;
    }

    public String getDescription() {
        return description;
    }

    public String getProducer() {
        return producer;
    }
}
```

One car component is its motor. As for all the other components, the motor has also got a producer, an identification number, and a description. In addition, a motor has also got a type: for instance, combustion engine, electric motor, or hybrid. Let's create the class `Motor` which inherits `Component`: a motor is a particular case of component.

```
public class Motor extends Component {

    private String motorType;

    public Motor(String motorType, String id, String producer, String description)
        super(id, producer, description);
        this.motorType = motorType;
    }
```

```
    public String getMotorType() {
        return motorType;
    }
}
```

The class definition `public class Motor extends Component` tells us that the class `Motor` inherits the functionality of `Component`. In the class `Motor`, we define the object variable `motorType`.

The Motor class constructor is interesting. In the first line of the constructor we notice the keyword `super`, which is used to call the superclass constructor. The call `super(id,producer,description)` class the constructor `public Component(String id, String producer, String description)` which is defined in the class `Component`; in this way the superclass object variables are assigned a value. After doing this, we assign a value to the object variable `motorType`.

When the class `Motor` inherits the class `Component`, in receives all the methods provides by `Component`. It is possible to create an instance of the class `Motor` as it is for any other class.

```
        Motor motor = new Motor("combustion engine", "hz", "volkswagen", "VW GOLF 1L 86-91");
        System.out.println(motor.getMotorType());
        System.out.println(motor.getProducer());
```

```
combustion engine
volkswagen
```

As you notice, the class `Motor` has the methods defined in `Component` at its disposal.

50.1 PRIVATE, PROTECTED AND PUBLIC

If a method or a variable have got the `private` field accessibility, it can not be seen by its subclasses, and its subclasses do not have any straight way to access it. In the previous example Motor can't access directly the attributes defined in its superclass Component (id, producer, description). The subclass see naturally everything which has been defined `public` in its super class. If we want to define superclass variables or methods whose accessibility should be restricted to only its subclasses, we can use the `protected` field accessability.

50.2 SUPERCLASS

The superclass constructor is defined by the `super` keyword. In fact, the call `super` is similar to the `this` constructor call. The call is given the values of the type required by the super class constructor parameter.

When we call the constructor, the variables defined in the super class are initialized. In fact, with constructor call happens the same thing as in normal constructor calls. Unless the superclass has a constructor without parameter, in the subclass constructor call there must always be a call for its superclass constructor.

Attention! The `super` call must always be in the first line!

50.3 CALLING THE SUPERCLASS METHODS

The method defined in the superclass can always be called using the `super` prefix, in the same way we call the methods defined in this class through the `this` prefix. For instance, we can make use of a method which overrives the superclass `toString` method in the following way:

```
@Override
public String toString() {
    return super.toString() + "\n  And my personal message again!";
}
```

Exercise 29: Person and their Heirs

EXERCISE 29.1: PERSON

Create the package `people` and the class `Person` in it; Person works in relation to the following main program:

```
public static void main(String[] args) {
    Person pekka = new Person("Pekka Mikkola", "Korsontie Street 1 03100 Vantaa");
    Person esko = new Person("Esko Ukkonen", "Mannerheimintie Street 15 00100 Helsinki"
    System.out.println(pekka);
    System.out.println(esko);
}
```

Printing

```
Pekka Mikkola
  Korsontie Street 1 03100 Vantaa
Esko Ukkonen
  Mannerheimintie Street 15 00100 Helsinki
```

EXERCISE 29.2: STUDENT

Create the class `Student` which *inherits* the class `Person`.

Students have 0 credits, at the beginning. As long as a student studies, their credits grow. Create the class, in relation to the following main program:

```
public static void main(String[] args) {
    Student olli = new Student("Olli", "Ida Albergintie Street 1 00400 Helsinki");
```

```
        System.out.println(olli);
        System.out.println("credits " + olli.credits());
        olli.study();
        System.out.println("credits "+ olli.credits());
    }
```

Prints:

```
Olli
  Ida Albergintie Street 1 00400 Helsinki
credits 0
credits 1
```

EXERCISE 29.3: TOSTRING FOR STUDETS

The `Student` in the previous exercise inherits their toString method from the class `Person`. Inherited methods can also be overwritten, that is to say replaced with another version. Create now an own version of the toString method for `Student`; the method has to work as shown below:

```
    public static void main(String[] args) {
        Student olli = new Student("Olli", "Ida Albergintie Street 1 00400 Helsinki");
        System.out.println( olli );
        olli.study();
        System.out.println( olli );
    }
```

Prints:

```
Olli
  Ida Albergintie Street 1 00400 Helsinki
  credits 0
Olli
  Ida Albergintie Street 1 00400 Helsinki
  credits 1
```

EXERCISE 29.4: TEACHER

Create the class `Teacher` in the same package. Teacher inherits `Person`, but they also have a salary, which together with the teacher information in String form.

Check whether the following main program generates the prinout below

```
    public static void main(String[] args) {
        Teacher pekka = new Teacher("Pekka Mikkola", "Korsontie Street 1 03100 Vantaa", 12
        Teacher esko = new Teacher("Esko Ukkonen", "Mannerheimintie 15 Street 00100 Helsinl
        System.out.println( pekka );
        System.out.println( esko );
```

```
        Student olli = new Student("Olli", "Ida Albergintie 1 Street 00400 Helsinki");
        for ( int i=0; i < 25; i++ ) {
            olli.study();
        }
        System.out.println( olli );
    }
```

Printing

```
Pekka Mikkola
  Korsontie Street 1 03100 Vantaa
  salary 1200 euros/month
Esko Ukkonen
  Mannerheimintie Street 15 00100 Helsinki
  salary 5400 euros/month
Olli
  Ida Albergintie Street 1 00400 Helsinki
  credits 25
```

EXERCISE 29.5: EVERYONE IN A LIST

Implement the method `public static void printDepartment(List<Person> people)` in the `Main` class, default package. The method prints all the people in the parameter list. When the `main` method is called, printDepartment should work in the following way.

```
    public static void printDepartment(List<Person> people) {
       // we print all the people in the department
    }

    public static void main(String[] args) {
        List<Person> people = new ArrayList<Person>();
        people.add( new Teacher("Pekka Mikkola", "Korsontie Street 1 03100 Vantaa", 1200)
        people.add( new Student("Olli", "Ida Albergintie Street 1 00400 Helsinki") );

        printDepartment(people);
    }
```

```
Pekka Mikkola
  Korsontie Street 1 03100 Vantaa
  salary 1200 euros/month
Olli
  Ida Albergintie Street 1 00400 Helsinki
  credits 0
```

50.4 THE OBJECT TYPE DEFINES THE CALLED METHOD: POLYMORPHISM

The method which can be called is defined through the variable type. For instance, if a `Student` object reference is saved into a `Person` variable, the object can use only the methods defined in the `Person` class:

```
Person olli = new Student("Olli", "Ida Albergintie Street 1 00400 Helsinki");
olli.credits();         // NOT WORKING!
olli.study();               // NOT WORKING!
String.out.println( olli );    // olli.toString() IT WORKS!
```

If the object has many different types, it has available the methods defined by *all* types. For instance, a `Student` object has available the methods defined both in the class `Person` and in `Object`.

In the previous exercise, we were in the class `Student` and we replaced the `toString` method inherited from `Person` with a new version of it. Suppose we are using an object throw a type which is not its real, what version of the object method would we call, then? For instance, below there are two students whose references are saved into a Person and an Object variables. We call the `toString` method of both. What version of the method is executed: the one defined in Object, in Person, or in Student?

```
Person olli = new Student("Olli", "Ida Albergintie Street 1 00400 Helsinki");
String.out.println( olli );

Object liisa = new Student("Liisa", "Väinö Auerin Street 20 00500 Helsinki");
String.out.println( liisa );
```

Printing:

```
Olli
  Ida Albergintie Street 1 00400 Helsinki
  credits 0
Liisa
  Väinö Auerin Street 20 00500 Helsinki
  credits 0
```

As you see, the execution method is chosen based on its real type, that is the type of the variable which saved the reference!

More generally: **The execution method is always chosen based on the object real type, regardless of the variable type which is used. Objects are diverse, which means they can be used through different variable types. The execution method does always depend of the object actual type.** This diversity is called polymorphysm.

50.5 ANOTHER EXAMPLE: POINTS

A point laying in a bidimensional coordinate system can be represented with the help of the following class:

```
public class Point {

    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int manhattanDistanceFromOrigin(){
        return Math.abs(x)+Math.abs(y);
    }

    protected String location(){
        return x + ", " + y;
    }

    @Override
    public String toString() {
        return "("+this.location()+") distance "+this.manhattanDistanceFromOrigin();
    }
}
```

The `location` method is not supposed to be used outside its class, and its accessability field is protected, which means only subclasses can access it. For instance, if we use a path fiding algorithm, the Manhattan distance is the distance of two points moving on a strictly horizontal and/or vertical path, along the coordinate system lines.

A coloured point is similar to a point, except that it contains a string which tells us its colour. The class can be created by inheriting Point:

```
public class ColouredPoint extends Point {

    private String colour;

    public ColouredPoint(int x, int y, String colour) {
        super(x, y);
        this.colour = colour;
    }

    @Override
    public String toString() {
        return super.toString()+" colour: "+colour;
    }
}
```

The class defines an object variable which saves the colour. The coordinates are saved in the superclass. The string representation must be similar to the one of the point, but it also has to show the colour. The overwritten `toString` method, calls the superclass toString method, and it adds the point colour to it.

In the following example, we create a list which contains various different points, either normal or coloured. Thanks to polymorphysm, we call the actual toString method of all objects, even though the list knows them as if they were all `Point`-type:

```
public class Main {
    public static void main(String[] args) {
        List<Point> points = new ArrayList<Point>();
        points.add(new Point(4, 8));
        points.add(new ColouredPoint(1, 1, "green"));
        points.add(new ColouredPoint(2, 5, "blue"));
        points.add(new Point(0, 0));

        for (Point point : points) {
            System.out.println(point);
        }
    }
}
```

Printing:

```
(4, 8) distance 12
(1, 1) distance 2 colour: green
(2, 5) distance 7 colour: blue
(0, 0) distance 0
```

We also want a 3D point in our program. Because that is not a coloured point, it shall inherit Point:

```
public class 3DPoint extends Point {

    private int z;

    public 3DPoint(int x, int y, int z) {
        super(x, y);
        this.z = z;
    }

    @Override
    protected String location() {
        return super.location() + ", " + z;     // printing as "x, y, z"
    }

    @Override
    public int manhattanDistanceFromOrigin() {
        // first, we ask the superclass for the distance of x+y
        // and then we add the value of z to it
```

```
        return super.manhattanDistanceFromOrigin() + Math.abs(z);
    }

    @Override
    public String toString() {
        return "(" + this.location() + ") distance " + this.manhattanDistanceFromOrigi
    }
}
```

A 3D point defines an object variable corresponding to the third coordinate, and it overrides the methods `location`, `manhattanDistanceFromOrigin` and `toString` so that they would take into account the tridimensionality. We can now extend the previous example and add 3D points to our list:

```
public class Main {

    public static void main(String[] args) {
        List<Point> points = new ArrayList<Point>();
        points.add(new Point(4, 8));
        points.add(new ColouredPoint(1, 1, "green"));
        points.add(new Point(2, 5, "blue"));
        points.add(new 3DPoint(5, 2, 8));
        points.add(new Point(0, 0));

        for (Point point : points) {
            System.out.println(point);
        }
    }
}
```

The output meets our expectations

```
(4, 8) distance 12
(1, 1) distance 2 colour: green
(2, 5) distance 7 colour: blue
(5, 2, 8) distance 15
(0, 0) distance 0
```

We notice that the tridimensional point `toString` method is exactly the same as the point's toString. Could we leave the toString method untouched? Of course! A tridimensional point can be reduced to the following:

```
public class 3DPoint extends Point {

    private int z;

    public 3DPoint(int x, int y, int z) {
        super(x, y);
        this.z = z;
    }

    @Override
```

```
    protected String distance() {
        return super.distance()+", "+z;
    }

    @Override
    public int manhattanDistanceFromOrigin() {
        return super.manhattanDistanceFromOrigin()+Math.abs(z);
    }
}
```

What does exactly happen when we call a tridimensional point's toString method? The execution proceeds in the following way:

- we look for a toString method in the class 3DPoint; this is not found and we move to its parent class
- we look for a toString method in the superclass Point; the method is found and we execute its code
- the code to execute is `return "("+this.location()+") location "+this.manhattanDistanceFromOrigin();`
- first, we execute the method location
- we look for a location method in the class 3DPoint; the method is found and we executes its code
- the location method calculates its result by calling the superclass method location
- next, we look for the definition of the method manhattanDistanceFromOrigin in the class Point3D; the method is found and we execute its code
- once again, the method calculates its result by calling its homonym in the superclass

The operating sequence produced by the method call has many steps. The idea is clear, anyway: when we want to execute a method, we first look for its definition in the object real type, and if it is not found, we move to the super class. If the method is not found in the parent class either, we move to the parent parent class, and so on...

50.6 WHEN DO WE HAVE TO USE INHERITANCE?

Inheritance is a tool to build and qualify object hierarchy; a subclass is always a special instance of the superclass. If the class we want to create is a special instance of an already existent class, we can create it by inheriting the class which already exists. For instance in our auto components example, motor *is* a component, but the motor has additional functionality which not all the classe s have.

Through inheritance, a subclass receives the superclass functionality. If a subclass does not need or use the inherited functionality, inheritance is not motivated. The inherited classes inherit the upperclass methods and interfaces, and therefore we can use a subclass for any purpose the superclass was used. It is good to stick to low inheritance hierarchy, because the more complex the inheritance hierarchy is, the more complex maintainance and further development will be. In general, if the hierarchy is higher than two or three, the program structure is usually to improve.

It is good to think about inheritance use. For instance, if `Car` inherited classes like `Component` or `Motor`, that would be wrong. A car *contains* a motor and components, but a car is not a motor or a

component. More generally, we can think that if an object owns or is composed of the other objects, inheritance is wrong.

Devoloping hierachy, you have to make sure the Single Responsibility Principle applies. There must be only one reason to change a class. If you notice that the hierarchy increases a class responsibilities, that class must be divided into various different classes.

50.6.1 AN EXAMPLE OF INHERITANCE MISUSE

Let's think about the `Customer` class, in relation to post service. The class contains the customer personal information, and `Order`, which inherits the customer personal information and contains the information of the object to order. The class `Order` has also got the method `mailingAddress`, which tells the mailing address of the order.

```
public class Customer {

    private String name;
    private String address;

    public Customer(String name, String address) {
        this.name = name;
        this.address = address;
    }

    public String getName() {
        return name;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }
}
```

```
public class Order extends Customer {

    private String product;
    private String amount;

    public Order(String product, String amount, String name, String address) {
        super(name, address);
        this.product = product;
        this.amount = amount;
    }

    public String getProduct() {
        return product;
    }
```

```
    public String getAmount() {
        return amount;
    }

    public String mailingAddress() {
        return this.getName() + "\n" + this.getAddress();
    }
}
```

The inheritance above is used erroneously. When a class inherits another, the subclass has to be a special instance of the superclass; order is not a special instance of customer. The misuse becomes apparent by breaking the single responsibility principle: the class `Order` is responsible for both the customer and the order information maintenance.

The problem with the previous solution becomes apparent when we think of what would happen if a customer changes their own address.

With a change of address, we would have to change *each* Order object of the customer, which is a hint about a bad situation. A better solution would be encapsulating `Customer` as an object variable of `Order`. If we think more specifically about the order semantics this becomes clear. *An order has a customer*. Let's change the class `Order` so that it contains a reference to `Customer`.

```
public class Order {

    private Customer customer;
    private String product;
    private String amount;

    public Tilaus(Customer customer, String product, String amount) {
        this.customer = customer;
        this.product = product;
        this.amount = amount;
    }

    public String getProduct() {
        return product;
    }

    public String getAmount() {
        return amount;
    }

    public String mailingAddress() {
        return this.customer.getName() + "\n" + this.customer.getAddress();
    }
}
```

The `Order` class above is better now. The method `mailingAddress` uses a *Customer* reference to retrieve the mailing address, instead of inheriting the class `Customer`. This makes easier both the maintanance and the concrete functionality of our program.

Now, when we modify a customer, we only need to change their information; we don't have to do anything about the orders.

Exercise 30: Container

Together with the exercise, you find the class `Container`, with the following constructor and methods:

- **public Container(double capacity)**
  It creates an empty container, whose capacity is given as argument; an improper capacity (<=0) creates a useless container, whose capacity is 0.
- **public double getVolume()**
  It returns the volume of product in the container.
- **public double getOriginalCapacity()**
  It returns the original capacity of the container, that is to say what the constructor was originally given.
- **public double getCurrentCapacity()**
  It returns the actual capacity of the container.
- **public void addToTheContainer(double amount)**
  It adds the specified amount of things to the container. If the amount is negative, nothing changes; if a part of that amount fits but not the whole of it, the container is filled up and the left over is thrown away.
- **public double takeFromTheContainer(double amount)**
  We take the specified amount form the container, the method returns what we **receive**. If the specified amount is negative, nothing happens and zero is returned. If we ask for more than what there is in the container, the method returns all the contents.
- **public String toString()**
  It returns the state of an object in String form like `volume = 64.5, free space 123.5`

In this exercise, we create various different containers out of our `Container` class. Attention! Create all the classes in the package `containers`.

EXERCISE 30.1: PRODUCT CONTAINER, PHASE 1

The class `Container` has control on the operations regarding the amount of a product. Now, we also want that products have their name and handling equipment. ***Program ProductContainer, a subclass of Container!*** Let's first implement a single object variable for the name of the contained product, a constructor and a getter for the name:

- **public ProductContainer(String productName, double capacity)**
  It creates an empty product container. The product name and the container capacity are given as parameters.
- **public String getName()**
  It returns the product name.

Remember in what way the constructor can make use of its upper class constructor in its first line!

Example:

```
ProductContainer juice = new ProductContainer("Juice", 1000.0);
juice.addToTheContainer(1000.0);
juice.takeFromTheContainer(11.3);
System.out.println(juice.getName()); // Juice
System.out.println(juice);           // volume = 988.7, free space 11.3
```

```
Juice
volume = 988.7, free space 11.3
```

EXERCISE 30.2: PRODUCT CONTAINER, PHASE 2

As you see from the example above, the `toString()` method inherited by ProductContainer does not know anything about the product name (of course!). *Something must be done for it!* Let's also add a setter for the product name, at the same time:

- **public void setName(String newName)** sets a new name to the product.
- **public String toString()** returns the object state in String form, like `Juice: volume = 64.5, free space 123.5`

The new `toString()` method could be programmed using the getter inherited from the superclass, retrieving the values of inherited but hidden values field values. However, we have programmed the superclass in a way that it is already able to produce the container situation in String form: why should we bother to program this again? Make use of the inherited `toString`.

Remember that an overwritten method can still be called in its subclass, where we overwrite it!

Use demonstration:

```
ProductContainer juice = new ProductContainer("Juice", 1000.0);
juice.addToTheContainer(1000.0);
juice.takeFromTheContainer(11.3);
System.out.println(juice.getName()); // Juice
juice.addToTheContainer(1.0);
System.out.println(juice);           // Juice: volume = 989.7, space 10.
```

```
Juice
Juice: volume = 989.7, free space 10.299999999999955
```

EXERCISE 30.3: CONTAINER HISTORY

Sometimes, it can be interesting to know in what way the container situation has changed: is the container often rather empty or full, is the fluctuation considerable or not, and so on. Let's provide our `ProductContainer` class with the ability to record the container history.

Let's start by designing a useful tool.

We could directly implement an `ArrayList<Double>` object to track our container history in the class *ProductConteiner*; however, now we create a *specific tool* for this purpose. The tool has to encapsulate an `ArrayList<Double>` object.

`ContainerHistory` public constructor and methods:

- **public ContainerHistory()** creates an empty `ContainerHistory` object.
- **public void add(double situation)** adds the parameter situation to the end of the container history.
- **public void reset()** it deletes the container history records.
- **public String toString()** returns the container history in the form of a String. *The String form given by the ArrayList class is fine and doesn't have to be modified.*

EXERCISE 30.4: CONTAINERHISTORY.JAVA, PHASE 2

Implement analysis methods for your `ContainerHistory` class:

- **public double maxValue()** reutrns the greatest value in the container history. If the history is empty, the method returns 0.
- **public double minValue()** reutrns the smallest value in the container history. If the history is empty, the method returns 0.
- **public double average()** reutrns the average of the values in the container history. If the history is empty, the method returns 0.

EXERCISE 30.5: CONTAINERHISTORY.JAVA, PHASE 3

Implement analysis methods for your `ContainerHistory` class:

- **public double greatestFluctuation()** returns the absolute value of the single greatest fluctuation in the container history (attention: a fluctuation of -5 is greater than 4). If the history is empty or it contains one value, the method returns zero. Absolute value is the distance of a number from zero. For instance the absolute value of -5.5 is 5.5, and the absolute value of 3.2 is 3.2.
- **public double variance()** returns the sample variance of the container history values. If the history is empty or it contains only one value, the method returns zero.

You find guidelines to calculate the variance in Wikipedia, in the population and sample variance section. For instance, the average of the numbers 3, 2, 7, and 2 is 3.5, and their sample variance is therefore ((3 - 3.5)² + (2 - 3.5)² + (7 - 3.5)² + (2 - 3.5)²)/(4 - 1) ˜ 5,666667.)

EXERCISE 30.6: PRODUCT CONTAINER RECORDER, PHASE 1

Implement the class `ProductContainerRecorder` which inherits `ProductContainer`. In addition to the old methods, the new version provides services for the container history. The hisotry is handled with a `ContainerHistory` object.

The public constructor and methods:

- **public ProductContainerRecorder(String productName, double capacity, double initialVolume)** creates a product container. The product name, capacity, and

original volume are given as parameter. *Record the original volume both as the stored product original volume and as the first value of the container history.*

- **public String history()** returns the container history in the following form: `[0.0, 119.2, 21.2]`. *Use the String printout form as it is.*

Attention: now we remember only the original volume.

Example:

```
// the well known way:
ProductContainerRecorder juice = new ProductContainerRecorder("Juice", 1000.0, 100
juice.takeFromTheContainer(11.3);
System.out.println(juice.getName()); // Juice
juice.addToTheContainer(1.0);
System.out.println(juice);           // Juice: volume = 989.7, free space 10.3
...
// history() does not work properly, yet:
System.out.println(juice.history()); // [1000.0]
   // in fact, we only retrieve the original value which was given to the constr
...
```

Printing:

```
Juice
Juice: volume = 989.7, free space 10.299999999999955
[1000.0]
```

EXERCISE 30.7: PRODUCT CONTAINER RECORDER, PHASE 2

It's time to pick up history! The first version of our history knew only the original value. Implement the following methods:

- **public void addToTheContainer(double amount)**; this works like the method in *Container*, but the new situation is recorded in the history. **Attention:** you have to record the product volume in the container after the addition, not the amount which was added!
- **public double takeFromTheContainer(double amount)**; it works like the method in `Container`, but the new situation is recorded in the history. **Attention:** you have to record the product volume in the container after the operation, not the amount which was removed!

Use example:

```
// the well known way:
ProductContainerRecorder juice = new ProductContainerRecorder("Juice", 1000.0, 100
juice.takeFromTheContainer(11.3);
System.out.println(juice.getName()); // Juice
juice.addToTheContainer(1.0);
System.out.println(juice);           // Juice: volume = 989.7, free space 10.3
...
// but now we have our history record
```

```
System.out.println(juice.history()); // [1000.0, 988.7, 989.7]
...
```

Printing:

```
Juice
Juice: volume = 989.7, free space 10.299999999999955
[1000.0, 988.7, 989.7]
```

Remember how an overwritten method can be used inside the method that overwrites it!

EXERCISE 30.8: PRODUCT CONTAINER RECORDER, PHASE 3

Implement the following method:

- **public void printAnalysis()**, which prints the history information regarding the product, following the exercise below:

Use example:

```
ProductContainerRecorder juice = new ProductContainerRecorder("Juice", 1000.0, 100
juice.takeFromTheContainer(11.3);
juice.addToTheContainer(1.0);
//System.out.println(juice.history()); // [1000.0, 988.7, 989.7]

juice.printAnalysis();
```

The method *printAnalysis* prints:

```
Product: Juice
History: [1000.0, 988.7, 989.7]
Greatest product amount: 1000.0
Smallest product amount: 988.7
Average: 992.8
Greatest change: 11.299999999999955
Variance: 39.129999999999676
```

EXERCISE 30.9: PRODUCT CONTAINER RECORDER, PHASE 4

Fill the analysis so that it prints the greatest fluctuation and the history variance.

50.7 INHERITANCE, INTERFACES, BOTH, OR NONE?

Inheritance does not exclude using interfaces, and viceversa. Interfaces are like an agreement on the class implementation, and they allow for the abstraction of the concrete implementation. Changing a class which implements an interface is quite easy.

As with interfaces, when we make use of inheritance, the subclasses are committed to provide all the superclass methods. Because of polymorphism, inheritance works as interfaces do. We can assign a subclass instance to a method which receives its superclass as parameter.

Below, we create a farm simulator, where we simulate the life in a farm. Note that the program does not make us of inheritance, and the interface use is scarce. With programs, we often create a first version which we improve later on. Typically, we don't already understand the scope of the problem when we implement the first version; planning interfaces and inheritance hierarchy may be difficult and it may slow down the work.

Exercise 31: Farm Simulator

Dairy farms have got milking animals; they do not handle milk themselves, but milk trucks transport it to dairy factories which process it into a variety of milk products. Each dairy factory is specialised in one product type; for instance, a cheese factory produces cheese, a butter factory produces butter, and a milk factory produces milk.

Let's create a simulator which represents the milk course of life. Implement all the classes in the package `farmsimulator`.

EXERCISE 31.1: BULK TANK

Milk has to be stored in bulk tanks in good conditions. Bulk tanks are produced both with a standard capacity of 2000 litres, and with customer specific capacity. Create the class BulkTank, with the following constructors and methods.

- **public BulkTank()**
- **public BulkTank(double capacity)**
- **public double getCapacity()**
- **public double getVolume()**
- **public double howMuchFreeSpace()**
- **public void addToTank(double amount)** adds to the tank only as much milk as it fits; the additional milk will not be added, and you don't have to worry about a situation where the milk spills over
- **public double getFromTank(double amount)** takes the required amount from the tank, or as much as there is left

Also, implement the `toString` method for your `BulkTank`. The `toString` method describes the tank situation by rounding down the litres using the `ceil()` method of class `Math`.

Test your bulk tank with the following program chunk:

```
BulkTank tank = new BulkTank();
tank.getFromTank(100);
tank.addToTank(25);
tank.getFromTank(5);
```

```
System.out.println(tank);

tank = new BulkTank(50);
tank.addToTank(100);
System.out.println(tank);
```

The program print output should look like the following:

```
20.0/2000.0
50.0/50.0
```

Note that when you call the `println()` method of the `out` object of class `System`, the method receives as paramater an `Object` variable; in such case, the print output is determined by the overwritten `toString()` method in `BulkTank`! We are in front of a case of polymorphism, because the method can work with different types.

EXERCISE 31.2: COW

If we want to produce milk, we also need cows. Cows have got names and udders. Udder capacity is a random value between 15 and 40; the class `Random` can be used to raffle off the numers, for instance, `int num = 15 + new Random().nextInt(26);`. The class `Cow` has the following functionality:

- **public Cow()** creates a new cow with a random name
- **public Cow(String name)** creates a new cow with its given name
- **String getName()** returns the cow's name
- **double getCapacity()** returns the udder capacity
- **double getAmount()** returns the amount on milk available in the cow's udders
- **String toString()** returns a String which describes the cow (see the example below)

`Cow` also implement the following interfaces: `Milkable`, which describes the cow's faculty for being milked, and `Alive`, which represents their faculty for being alive.

```
public interface Milkable {
    public double milk();
}

public interface Alive {
    public void liveHour();
}
```

When a cow is milked, all their milk provision is taken to be processed. As long as a cow lives, their milk provision increases slowly. In Finland, milking cows produce 25-30 litres of milk every day, on the average. We simulate this by producing 0.7-2 litres every hour.

If a cow is not given a name, they are assigned a random one from the list below.

```
    private static final String[] NAMES = new String[]{
        "Anu", "Arpa", "Essi", "Heluna", "Hely",
        "Hento", "Hilke", "Hilsu", "Hymy", "Ihq", "Ilme", "Ilo",
```

```
            "Jaana", "Jami", "Jatta", "Laku", "Liekki",
            "Mainikki", "Mella", "Mimmi", "Naatti",
            "Nina", "Nyytti", "Papu", "Pullukka", "Pulu",
            "Rima", "Soma", "Sylkki", "Valpu", "Virpi"};
```

Implement the class Cow, and test whether it works with the following program body.

```
Cow cow = new Cow();
System.out.println(cow);


Alive livingCow = cow;
livingCow.liveHour();
livingCow.liveHour();
livingCow.liveHour();
livingCow.liveHour();

System.out.println(cow);

Milkable milkingCow = cow;
milkingCow.milk();

System.out.println(cow);
System.out.println("");

cow = new Cow("Ammu");
System.out.println(cow);
cow.liveHour();
cow.liveHour();
System.out.println(cow);
cow.milk();
System.out.println(cow);
```

The program print output can be like the following.

```
Liekki 0.0/23.0
Liekki 7.0/23.0
Liekki 0.0/23.0
Ammu 0.0/35.0
Ammu 9.0/35.0
Ammu 0.0/35.0
```

EXERCISE 31.3: MILKINGROBOT

In modern dairy farms, milking robots handle the milking. The milking robot has to be connected to the bulk tank in order to milk an udder:

- **public MilkingRobot()** creates a new milking robot
- **BulkTank getBulkTank()** returns the connected bulk tank, or a `null` reference, if the tank hasn't been installed

- **void setBulkTank(BulkTank tank)** installs the parameter bulk tank to the milking robot
- **void milk(Milkable milkable)** milks the cow and fills the connected bulk tank; the method returns an `IllegalStateException` is no tank has been fixed

Implement the class MilkingRobot, and test it using the following program body. Make sure that the milking robot can milk all the objects which implement the interface Milkable!

```
        MilkingRobot milkingRobot = new MilkingRobot();
        Cow cow = new Cow();
        milkingRobot.milk(cow);
```

```
Exception in thread "main" java.lang.IllegalStateException: The MilkingRobot hasn
        at farmsimulator.MilkingRobot.milk(MilkingRobot.java:17)
        at farmsimulator.Main.main(Main.java:9)
Java Result: 1
```

```
MilkingRobot milkingRobot = new MilkingRobot();
Cow cow = new Cow();
System.out.println("");

BulkTank tank = new BulkTank();
milkingRobot.setBulkTank(tank);
System.out.println("Bulk tank: " + tank);

for(int i = 0; i < 2; i++) {
    System.out.println(cow);
    System.out.println("Living..");
    for(int j = 0; j < 5; j++) {
        cow.liveHour();
    }
    System.out.println(cow);

    System.out.println("Milking...");
    milkingRobot.milk(cow);
    System.out.println("Bulk tank: " + tank);
    System.out.println("");
}
```

The print output of the program can look like the following:

```
Bulk tank: 0.0/2000.0
Mella 0.0/23.0
Living..
Mella 6.2/23.0
Milking...
Bulk tank: 6.2/2000.0

Mella 0.0/23.0
```

```
Living..
Mella 7.8/23.0
Milking...
Bulk tank: 14.0/2000.0
```

EXERCISE 31.4: BARN

Cows are kept (and in this case milked) in barns. The original barns have room for one milking robot. Note that when milking robots are installed, they are connected to a specific barn's bulk tank. If a barn does not have a milking robot, it can't be used to handle the cow, either. Implement the class `Barn` with the following constructor and methods:

- **public Barn(BulkTank tank)**
- **public BulkTank getBulkTank()** returns the barn's bulk tank
- **public void installMilkingRobot(MilkingRobot milkingRobot)** installs a milking robot and connects it to the barn bulk tank
- **public void takeCareOf(Cow cow)** milks the parameter cow with the help of the milking robot, the method throws an `IllegalStateException` if the milking robot hasn't been installed
- **public void takeCareOf(Collection<Cow> cows)** milks the parameter cows with the help of the milking robot, the method throws an `IllegalStateException` if the milking robot hasn't been installed
- **public String toString()** returns the state of the bulk tank contained by the barn

`Collection` is Java's own interface, and it represents collections' behaviour. For instance, the classes `ArrayList` and `LinkedList` implement the interface `Collection`. All instances of classes which implement `Collection` can be iterated with a for-each construction.

Test your class `Barn` with the help of the following program body. Do not pay to much attention to the class `LinkedList`; apparently, it works as `ArrayList`, but the implemantation in encapsulates is slightly different. More information about this in the data structures course!

```
Barn barn = new Barn(new BulkTank());
System.out.println("Barn: " + barn);

MilkingRobot robot = new MilkingRobot();
barn.installMilkingRobot(robot);

Cow ammu = new Cow();
ammu.liveHour();
ammu.liveHour();

barn.takeCareOf(ammu);
System.out.println("Barn: " + barn);

LinkedList<Cow> cowList = new LinkedList<Cow>();
cowList.add(ammu);
cowList.add(new Cow());
```

```
for(Cow cow: cowList) {
    cow.liveHour();
    cow.liveHour();
}

barn.takeCareOf(cowList);
System.out.println("Barn: " + barn);
```

The print output should look like the following:

```
Barn: 0.0/2000.0
Barn: 2.8/2000.0
Barn: 9.6/2000.0
```

EXERCISE 31.5: FARM

Farms have got an owner, a barn and a herd of cows. Farm also implements our old interface `Alive`: calling the method `liveHour` makes all the cows of the farm live for an hour. You also have to create method `manageCows` which calls Barn's method `takeCareOf` so that all cows are milked. Implement your class Farm, and make it work according to the following example.

```
Farm farm = new Farm("Esko", new Barn(new BulkTank()));
System.out.println(farm);

System.out.println(farm.getOwner() + " is a tough guy!");
```

Expected print output:

```
Farm owner: Esko
Barn bulk tank: 0.0/2000.0
No cows.
Esko is a tough guy!
```

```
Farm farm = new Farm("Esko", new Barn(new BulkTank()));
farm.addCow(new Cow());
farm.addCow(new Cow());
farm.addCow(new Cow());
System.out.println(farm);
```

Expected print output:

```
Farm owner: Esko
Barn bulk tank: 0.0/2000.0
Animals:
        Naatti 0.0/19.0
        Hilke 0.0/30.0
        Sylkki 0.0/29.0
```

```
Farm farm = new Farm("Esko", new Barn(new BulkTank()));

farm.addCow(new Cow());
farm.addCow(new Cow());
farm.addCow(new Cow());

farm.liveHour();
farm.liveHour();
System.out.println(farm);
```

Expected print output:

```
Farm owner: Esko
Barn bulk tank: 0.0/2000.0
Animals:
        Heluna 2.0/17.0
        Rima 3.0/32.0
        Ilo 3.0/25.0
```

```
Farm farm = new Farm("Esko", new Barn(new BulkTank()));
MilkingRobot robot = new MilkingRobot();
farm.installMilkingRobot(robot);

farm.addCow(new Cow());
farm.addCow(new Cow());
farm.addCow(new Cow());


farm.liveHour();
farm.liveHour();

farm.manageCows();

System.out.println(farm);
```

Expected print output:

```
Farm owner: Esko
Barn bulk tank: 18.0/2000.0
Animals:
        Hilke 0.0/30.0
        Sylkki 0.0/35.0
        Hento 0.0/34.0
```

50.8 AN ABSTRACT CLASS

Abstract classes combine interfaces and inheritance. They do not produce instances, but you can create instances of their subclasses. An abstract class can contain both normal and abstract methods, the first containing the method body, the second having only the method definition. The implementation of the abstract methods is left to the inheriting class. In general, we use abstract classes when the object they represent is not a clear, self-defined concept. In such cases, it is not possible to create instances of it.

Both when we define abstract classes and abstract methods, we use the keyword `abstract`. An abstract class is defined by the statement `public abstract class ClassName`, whereas an abstract method is defined by `public abstract returnType methodName`. Let's consider the following abstract class `Operation`, which provides a framework for operations, and their excecutions.

```
public abstract class Operation {

    private String name;

    public Operation(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

    public abstract void execute(Scanner reader);
}
```

The abstract class `Operation` works as a framework to excetute different operations. For instance, an addition can be implemented by inheriting the class `Operation` in the following way.

```
public class Addition extends Operation {

    public Addition() {
        super("Addition");
    }

    @Override
    public void execute(Scanner reader) {
        System.out.print("Give the first number: ");
        int first = Integer.parseInt(reader.nextLine());
        System.out.print("Give the second number: ");
        int second = Integer.parseInt(reader.nextLine());

        System.out.println("The sum is " + (first + second));
    }
}
```

Because all classes which descend from `Operation` are also Operation-type, we can create a user interface based on `Operation`-type variables. The following class `UserInterface` contains a list of operations and a reader. The operations can be added dynamically in the user interface.

```
public class UserInterface {

    private Scanner reader;
    private List<Operation> operations;

    public UserInterface(Scanner reader) {
        this.reader = reader;
        this.operations = new ArrayList<Operation>();
    }

    public void addOperation(Operation operation) {
        this.operations.add(operation);
    }

    public void start() {
        while (true) {
            printOperations();
            System.out.println("Choice: ");

            String choice = this.reader.nextLine();
            if (choice.equals("0")) {
                break;
            }

            executeOperation(choice);
            System.out.println();
        }
    }

    private void printOperations() {
        System.out.println("\t0: Quit");
        for (int i = 0; i < this.operations.size(); i++) {
            String operationName = this.operations.get(i).getName();
            System.out.println("\t" + (i + 1) + ": " + operationName);
        }
    }

    private void executeOperation(String choice) {
        int operation = Integer.parseInt(choice);

        Operation chosen = this.operations.get(operation - 1);
        chosen.execute(reader);
    }
}
```

The user interface works in the following way:

```
        UserInterface ui = new UserInterface(new Scanner(System.in));
        ui.addOperation(new Addition());

        ui.start();
```

```
Operations:
        0: Quit
        1: Addition
Choice: 1
Give the first number: 8
Give the second number: 12
The sum is 20

Operations:
        0: Quit
        1: Addition
Choice: 0
```

The difference between interfaces and abstract classes is that abstract classes provide the program with more structure. Because it is possible to define the functionality of abstract classes, we can use them to define the default implementation, for instance. The user interface above made use of a definition of the abstract class to store the operation name.

Exercise 32: Different Boxes

Together with the exercise body, you find the classes `Thing` and `Box`. The class `Box` is abstract, and it is programmed so that adding things always implies calling the method `add`. The `add` method, resposible of adding one thing, is abstract, and any box which inherits the class `Box` has to implement the method `add`. Your task is modifying the class `Thing` and implementing various different boxes based on `Box`.

Add all new classes to the package `boxes`.

```
package boxes;

import java.util.Collection;

public abstract class Box {

    public abstract void add(Thing thing);

    public void add(Collection<Thing> things) {
        for (Thing thing : things) {
            add(thing);
        }
    }
```

```
    public abstract boolean isInTheBox(Thing thing);
}
```

EXERCISE 32.1: MODIFICATIONS TO THING

Add an inspection to the constructor of `Thing`, to make sure that the thing's weight is never negative (weight 0 is accepted). If the weight is negative, the constructor has to throw an `IllegalArgumentException`. Also implement the methods `equals` and `hashCode` in the class `Thing`, allowing you to use the `contains` method of different lists and collections. Implement the methods without taking into consideration the value of the object variable `weight`. *Of course, you can use NetBeans functionality to implement equals and hashCode.*

EXERCISE 32.2: MAXIMUM WEIGHT BOX

Implement the class `MaxWeightBox` in the package `boxes`; the class inherits `Box`. MaxWeightBox has the constructor `public MaxWeightBox(int maxWeight)`, which determines the box maximum weight. Things can be added to MaxWeightBox if and only if the thing weight does not exceed the box weight.

```
        MaxWeightBox coffeeBox = new MaxWeightBox(10);
        coffeeBox.add(new Thing("Saludo", 5));
        coffeeBox.add(new Thing("Pirkka", 5));
        coffeeBox.add(new Thing("Kopi Luwak", 5));

        System.out.println(coffeeBox.isInTheBox(new Thing("Saludo")));
        System.out.println(coffeeBox.isInTheBox(new Thing("Pirkka")));
        System.out.println(coffeeBox.isInTheBox(new Thing("Kopi Luwak")));
```

```
true
true
false
```

EXERCISE 32.3: ONE-THING BOX AND BLACK-HOLE BOX

Next, implement the class `OneThingBox` in the package `boxes`; the class inherits `Box`. OneThingBox has the constructor `public OneThingBox()`, and only one thing can fit there. If the box already contains one thing, this should not be changed. The weight of the added thing is not important.

```
        OneThingBox box = new OneThingBox();
        box.add(new Thing("Saludo", 5));
        box.add(new Thing("Pirkka", 5));

        System.out.println(box.isInTheBox(new Thing("Saludo")));
        System.out.println(box.isInTheBox(new Thing("Pirkka")));
```

```
true
false
```

Next, implement the class `BlackHoleBox` in the package `boxes`; the class inherits `Box`. BlackHoleBox has the constructor `public BlackHoleBox()`; any thing can be added to a black-hole box, but none will be found when you'll look for them. In other words, adding things must always work, but the method `isInTheBox` has to return always false.

```
        BlackHoleBox box = new BlackHoleBox();
        box.add(new Thing("Saludo", 5));
        box.add(new Thing("Pirkka", 5));

        System.out.println(box.isInTheBox(new Thing("Saludo")));
        System.out.println(box.isInTheBox(new Thing("Pirkka")));
```

```
false
false
```

50.9 REMOVING OBJECTS FROM AN ARRAYLIST

In the following exercise we see what you may end up to, when you want to remove a part of the list objects while parsing an ArrayList:

```
// somewhere, with a definition like:
// ArrayList<Object> list = new ...

for ( Object object : list ) {
   if ( hasToBeRemoved(object) ) {
      list.remove(object);
   }
}
```

The solution does not work and it throws a `ConcurrentModificationException`, because it is not possible to modify a list while parsing it with a *foreach* iterator. We will come back to the topic better on week 12. If you run into such a situation, you can handle it in the following way:

```
// somewhere, with a definition like:
// ArrayList<Object> list = new ...

ArrayList<Object> toBeRemoved = new ArrayList<Object>();

for ( Object object : list ) {
   if ( hasToBeRemoved(object) ) {
```

```
            toBeRemoved.add(object);
        }
    }

    list.removeAll(toBeRemoved);
```

The objects which have to be deleted are gathered together while we parse the list, and the remove operation is executed only after parsing the list.

Exercise 33: Dungeon

This exercise is worth four points. Attention! Implement all the functionality in the package `dungeon`.

Attention: you can create only one Scanner object to make your tests work. Do not use Scandinavian letters in the class names. Also, do not use static variables, the tests execute your program many different times, and the static variable values left from the previous execution would possibly disturb them!

In this exercise, you implement a dungeon game. In the game, the player is in a dungeon full of vampires. The player has to destroy the vampires before his lamp runs out of battery and the vampires can suck his blood in the darkness. The player can see the vampires with a blinking of their lamp, after which they have to move blind before the following blinking. With one move, the player can walk as many steps as they want.

The game situation, i.e. the dungeon, the player and the vampires are shown in text form. The first line in the print output tells how many moves the player has left (that is to say, how much battery the lamp has). After that, the print output shows player and vampire positions, which in turn are followed by the game map. In the example below, you see the player (`@`) and three vampires (`v`); in this case, the player has enough light for fourteen moves.

```
14

@ 1 2
v 6 1
v 7 3
v 12 2

..................
......v...........
.@.........v......
.......v..........
```

The example above shows the lamp has enough battery for 14 blinkings. The player `@` is located at `1 2`. Note that the coordinates are calculated starting from the high left corner of the game board. In the map below, the character `x` is located at `0 0`, `y` is at `2 0` and `z` is at `0 2`.

```
X.Y..............
................
Z...............
................
```

The user can move by giving a sequence of commands and pressing Enter. The commands are:

- `w` go up
- `s` go down
- `a` go left
- `d` go right

When the user commands are executed (the user can give many commands at once), a new game situation is drawn. If the lamp charge reaches 0, the game ends and the text `YOU LOSE` is printed on the board.

The vampires move randomly in the game, and they take one step for each step the player takes. If the player and a vampire run into each other (even momentarily) the vampire is destroyed. If a vampire tries to step outside the board, or into a place already occupied by another vampire, the move is not executed. When all the vampires are destroyed, the game ends and it prints `YOU WIN`.

In order to help the tests, create the class `Dungeon` in your game, with:

- the constructor **`public Dungeon(int length, int height, int vampires, int moves, boolean vampiresMove)`**

  the values `length` and `height` represent the dimension of the dungeon (always a rectangle); `vampires` stands for the initial number of vampires (the positions of the vampires can be decided randomly); `moves` determines the initial number of moves; and if `vampiresMove` is `false`, the vampires do not move.

- the method **`public void run()`**, which starts the game

Attention! The player starts the game in the position 0,0!

Attention! Player and vampires can not move out of the dungeon and two vampires cannot step into the same place!

Below, you find a couple of examples to help you to understand the situation better:

```
14

@ 0 0
v 1 2
v 7 8
v 7 5
v 8 0
v 2 9

@.......v.
..........
```

```
.v........
..........
..........
.......v..
..........
..........
.......v..
..v.......

ssd
13

@ 1 2
v 8 8
v 7 4
v 8 3
v 1 8

..........
..........
.@........
........v.
.......v..
..........
..........
..........
.v......v.
..........

ssss
12

@ 1 6
v 6 9
v 6 5
v 8 3

..........
..........
..........
........v.
..........
......v...
.@........
..........
..........
......v...

dd
11

@ 3 6
v 5 9
v 6 7
```

```
v 8 1

..........
........v.
..........
..........
..........
..........
...@......
......v...
..........
.....v....

ddds
10

@ 6 7
v 6 6
v 5 0

.....v....
..........
..........
..........
..........
..........
......v...
......@...
..........
..........

w
9

@ 6 6
v 4 0

....v.....
..........
..........
..........
..........
..........
......@...
..........
..........
..........

www
8

@ 6 3
v 4 0
```

```
....v.....
..........
..........
......@...
..........
..........
..........
..........
..........
..........

aa
7

@ 4 3
v 4 2

..........
..........
....v.....
....@.....
..........
..........
..........
..........
..........
..........

w
YOU WIN
```