Part 9

Object polymorphism

Learning Objectives

- You are familiar with the concept of inheritance hierarchy.
- You understand that an object can be represented through all of its actual types.

We've encountered situations where reference-type variables have other types besides their own one. For example, *all* objects are of type `Object`, i.e., any given object can be represented as a `Object`-type variable in addition to its own type.

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

```
String text = "text";
Object textString = text;
```

In the examples above, a string variable is represented as both a String type and an Object type. Also, a String-type variable is assigned to an Object-type variable. However, assignment in the other direction, i.e., setting an Object-type variable to a String type, will not work. This is because `Object`-type variables are not of type `String`

```
Object textString = "another string";
String text = textString; // WON'T WORK!
```

What is this all about?

In addition to each variable's original type, each variable can also be represented by the types of interfaces it implements and classes that it inherits. The String class inherits the Object class and, as such, String objects are always of type Object. The Object class does not inherit a String class, so Object-type variables are not automatically of type String. Take a closer look at the String API documentation, in particular at the top of the HTML page.

The API documentation for the String class begins with a generic header followed by the class' package (`java.lang`). After the package details, the name of the class (`Class String`) is followed by the *inheritance hierarchy* of the class.

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

The inheritance hierarchy lists all the classes that the given class has inherited. Inherited classes are listed in the order of inheritance, with class being inspected always at the bottom. In the inheritance hierarchy of the String class, we see that the `String` class inherits the `Object` class. *In Java, each class can inherit one class at most*. On the other hand, the inherited class may have inherited another class. As such, a class may indirectly inherit more than a single class.

The inheritance hierarchy can also be thought of as a list of the different types that the class implements.

Knowledge of the fact that objects can be of many different types — of type Object, for instance — makes programming simpler. If we only need methods defined in the Object class, such as `toString`, `equals` and `hashCode` in a method, we can simply use `Object` as the type of the

method parameter. In that case, you can pass the method for *any* object as a parameter. Let's take a look at this with the `printManyTimes` method. The method gets an `Object`-type variable and the number of print operations as its parameters.

```
public class Printer {

    public void printManyTimes(Object object, int times) {
        int i = 0;
        while (i < times) {
            System.out.println(object.toString());
            // or System.out.println(object);

            i = i + 1;
        }
    }
}
```

The method can be given any type of object as a parameter. Within the `printManyTimes` method, the object only has access to the methods defined in the `Object` class because the object is *known* in the method to be of type `Object`. The object may, in fact, be of another type.

```
Printer printer = new Printer();

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

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

Sample output

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

Let's continue to look at the API description of the `String` class. The inheritance hierarchy in the description is followed by a list of interfaces implemented by the class.

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

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

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

Since we're able to define the type of a method's parameter, we can declare methods that receive an object that *implements a specific interface*. When a method's parameter is an interface, any object that implements that interface can be passed to it as an argument.

We'll extend the `Printer` class so that it has a method for printing the characters of objects that implement the `CharSequence` interface. The `CharSequence` interface provides, among other things, methods `int length()` for getting a string's length and `char charAt(int index)`, which retrieves a character from a given index.

```
public class Printer {

    public void printManyTimes(Object object, int times) {
        int i = 0;
        while (i < times) {
            System.out.println(object);
            i = i + 1;
        }
    }

    public void printCharacters(CharSequence charSequence) {
        int i = 0;
        while (i < charSequence.length()) {
            System.out.println(charSequence.charAt(i));
            i = i + 1;
        }
```

```
    }
}
```

The `printCharacters` method can be passed any object that implements the `CharSequence` interface. These include `String` as well as `StringBuilder`, which is often more functional for building strings than `String`. The `printCharacters` method prints each character of a given object on its own line.

```
Printer printer = new Printer();

String string = "works";

printer.printCharacters(string);
```

Sample output

```
w
o
r
k
s
```

Programming exercise:

Herds (2 points)

Points 2/2

NB! By submitting a solution to a part of an exercise which has multiple parts, you can get part of the exercise points. You can submit a part by using the 'submit' button on NetBeans. More on the programming exercise submission instructions: Exercise submission instructions.

In this exercise we are going to create organisms and herds of organisms that can move around. To represent the locations of the organisms we'll use a **two-dimensional coordinate system**. Each position involves two numbers: `x` and `y` coordinates. The `x` coordinate indicates how far from the origin (i.e. point zero, where x = 0, y = 0) that position is horizontally. The `y` coordinate indicates the

distance from the origin vertically. If you are not familiar with using a coordinate system, you can study the basics from Wikipedia.

The exercise base includes the interface `Movable`, which represents something that can be moved from one position to another. The interface includes the method `void move(int dx, int dy)`. The parameter `dx` tells how much the object moves on the x axis, and dy tells the distance on the y axis.

This exercise involves implementing the classes `Organism` and `Herd`, both of which are movable.

Part 1: Implementing the Organism Class

Create a class called `Organism` that implements the interface `Movable`. An organism should know its own location (as x, y coordinates). The API for the class `Organism` is to be as follows:

- **public Organism(int x, int y)**
  The class constructor that receives the x and y coordinates of the initial position as its parameters.
- **public String toString()**
  Creates and returns a string representation of the organism. That representation should remind the following: `"x: 3; y: 6"`. Notice that a semicolon is used to separate the coordinates.
- **public void move(int dx, int dy)**
  Moves the object by the values it receives as parameters. The `dx` variable contains the change to coordinate `x`, and the `dy` variable ontains the change to the coordinate `y`. For example, if the value of `dx` is 5, the value of the object variable `x` should be incremented by five.

Use the following code snippet to test the `Organism` class.

```
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);
```

Sample output

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

Part 2: Implementing the Herd

Create a class called `Herd` that implements the interface `Movable`. A herd consists of multiple objects that implement the Movable interface. They must be stored in e.g. a list data structure.

The `Herd` class must have the following API.

- **public String toString()**
  Returns a string representation of the positions of the members of the herd, each on its own line.
- **public void addToHerd(Movable movable)**
  Adds an object that implements the `Movable` interface to the herd.
- **public void move(int dx, int dy)**
  Moves the herd with by the amount specified by the parameters. Notice that here you have to move each member of the herd.

Test out your program with the sample code below:

```
Herd herd = new Herd();
herd.addToHerd(new Organism(57, 66));
herd.addToHerd(new Organism(73, 56));
herd.addToHerd(new Organism(46, 52));
herd.addToHerd(new Organism(19, 107));
System.out.println(herd);
```

Sample output

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

Exercise submission instructions

How to see the solution

Programming exercise:
Animals (4 parts)

Points
4/4

NB! By submitting a solution to a part of an exercise which has multiple parts, you can get part of the exercise points. You can submit a part by using the 'submit' button on NetBeans. More on the programming exercise submission instructions: Exercise submission instructions.

In this exercise you'll demonstrate how to use inheritance and interfaces.

Part 1: Animal

First implement an abstract class called `Animal`. The class should have a constructor that takes the animal's name as a parameter. The Animal class also has non-parameterized methods eat and sleep that return nothing (void), and a non-parameterized method getName that returns the name of the animal.

The sleep method should print "(name) sleeps", and the eat method should print "(name) eats". Here (name) is the name of the animal in question.

Part 2: Dog

Implement a class called `Dog` that inherits from Animal. Dog should have a parameterized constructor that can be used to name it. The class should also have a non-parameterized constructor, which gives the dog the name "Dog". Another method that Dog must have is the non-parameterized bark, and it should not return any value (void). Like all animals, Dog needs to have the methods eat and sleep.

Below is an example of how the class Dog is expected to work.

```
Dog dog = new Dog();
dog.bark();
dog.eat();

Dog fido = new Dog("Fido");
fido.bark();
```

Sample output

```
Dog barks
Dog eats
Fido barks
```

Part 3: Cat

Next to implement is the class `Cat`, that also inherits from the Animal class. Cat should have two constructors: one with a parameter, used to name the cat according to the parameter, and one without parameters, in which case the name is simply "Cat". Another method for Cat is a non-parameterized method called purr that returns no value (void). Cats should be able to eat and sleep like in the first part.

Here's an example of how the class Cat is expected to function:

```
Cat cat = new Cat();
cat.purr();
cat.eat();

Cat garfield = new Cat("Garfield");
garfield.purr();
```

Sample output

```
Cat purrs
Cat eats
Garfield purrs
```

Part 4: NoiseCapable

Finally, create an interface called `NoiseCapable`. It should define a non-parameterized method makeNoise that returns no value (void). Implement the interface in the classes Dog and Cat. The interface should take use of the bark and purr methods you've defined earlier.

Below is an example of the expected functionality.

```
NoiseCapable dog = new Dog();
dog.makeNoise();

NoiseCapable cat = new Cat("Garfield");
cat.makeNoise();
Cat c = (Cat) cat;
c.purr();
```

Sample output

```
Dog barks
Garfield purrs
Garfield purrs
```

Exercise submission instructions

How to see the solution

You have reached the end of this section! Continue to the next section:

→ 4. Summary

Remember to check your points from the ball on the bottom-right corner of the material!

In this part:

1. Class inheritance
2. Interfaces

Source code of the material

This course is created by the Agile Education Research -research group of the University of Helsinki.

Credits and about the material.



HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

MASSIIVISET AVOIMET VERKKOKURSSIT
MASSIVE OPEN ONLINE COURSES · MOOC.FI