Hyperiondev

TASK

Object-Oriented Programming — Inheritance, Interfaces and Static Methods

Visit our website

Introduction

WELCOME TO THE INHERITANCE, INTERFACES AND STATIC METHODS TASK!

In this task, you will learn three vital elements of Object-Oriented Programming: inheritance, interfaces and static methods. You will need to be comfortable with these elements for your projects to come.

Get in touch

Connect for support

Remember that with our courses, you're not alone! You can contact an expert code reviewer to get support on any aspect of your course.

The best way to get help is to login to Discord at **https://discord.com/invite/hyperdev** where our specialist team is ready to support you.

Our team is happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

WHAT IS INHERITANCE?

Inheritance is one of the features of Object-Oriented Programming (OOP). Inheritance allows a class to use (or inherit) the properties and methods of another class. In other words, the **derived class** inherits the states and behaviours from the **base class**. The derived class is also called the **subclass**, and the base class is also known as the **superclass**. The derived class can add its own additional variables and methods. These additional variables and methods differentiate the derived class from the base class.

Inheritance is a compile-time mechanism. A superclass can have any number of subclasses. But a subclass can have only one superclass. This is because Java does not support **multiple inheritances**.

To help us understand inheritance, we are going to consider an example using animals. Consider two animals – a lion and a cheetah. Although these are not the same animal, they are both animals, so they have certain things in common. Let's say that all animals have four legs and a head and teeth. Because all animals have these features, we can define a lion and a cheetah as animals.

Now let's put this in a programming context. If we had to create a class for a lion and one for a cheetah, we would find that we would be duplicating a lot of the same methods and fields in the two classes because the two animals are similar and therefore have features in common. Programmers are lazy! So we always want to try to do things as efficiently as possible. Therefore, we minimise the number of duplicates we create. To do this in this example, we could make use of inheritance. Start by creating a class for animals in which we define all the common methods and fields for the two animals. Create another two classes – one for the lion and one for the cheetah. These would both inherit all the methods and fields from the **superclass** – the animal class. This means the lion and cheetah classes would both have all the properties that the animal class has as well as others, which would be specific to that class. Think of the lion and cheetah class as subclasses of the animal class. They are **derived** from the animal class.

INHERITANCE — HANDS ON!

Let's create the scenario above. Create an animal class with variables common to both the lion and the cheetah, e.g. number of teeth, spots (boolean), weight. We will now create the lion class. An example animal class has been included in the task folder in case you get confused.

It is, however, advised that you create one on your own to practise your skills. The *Animal* class is very similar to previous classes you have created, so there should be nothing new there. The *Lion* class will seem quite different from any other class you have created. Do not worry, you will understand everything shortly.

Below is the initial setup of the `Lion` class:

```
public class Lion extends Animal {
   public Lion(int numTeeth, boolean spots, int weight) {
      super(numTeeth, spots, weight);
   }
}
```

You will notice many unusual things here. The first will be the word `extends`. This is a Java keyword which tells Java that you are using inheritance. It also tells Java that the class name which follows this keyword is the class you will be inheriting from – in this case it is the `Animal` class.

The next thing you will notice is the constructor. If the class you are inheriting from has a constructor which requires parameters, these parameters have to be passed into the subclass. The superclass object must be constructed using these parameters.

You will see that the keyword `super` is used to denote the superclass. The constructor of the superclass is used here without defining a specific object for it. This is a special case because it is being used in inheritance and therefore it will not cause an error. When accessing methods and variables of the superclass, the `super` keyword will be used along with the dot (`.`) operator.

You can now add anything to this class that is specific to a `Lion`. It will have those features in addition to those which are specific to all animals.

Please note that when you use a field or a method of the superclass, it will not change for all instances of the superclass. It will only change for that particular instance. Inheritance is not the same as using static variables or methods.

INTRODUCTION TO INTERFACES

An interface is similar to a class. However, unlike classes, it can only contain method signatures and constants. A method signature is simply the name and the parameter types of a method, and not its implementation. An interface can be

thought of as a blueprint of a class: they specify what a class must do but not how to go about doing it.

Interfaces are used to achieve full abstraction. Abstraction involves showing only the relevant data and hiding all unnecessary details of an object from the user. The methods in an interface do not contain bodies, they have to be implemented by a class before you can access them. The class that implements an interface must implement all of its methods.

Java uses the **`interface`** keyword to define an interface. The syntax used to define an interface is as follows:

```
public interface InterfaceName {
   //Constant declarations
   // Method signatures
}
```

Below is an example of an interface called Edible that that includes a single method named **`howToEat()`**:

```
public interface Edible {
   // Describes how to eat
   public abstract String howToEat();
}
```

You can use this interface to specify whether an object is edible. This is accomplished by letting the class for the object implement this interface using the **`implements`** keyword. The name of an interface is an adjective (such as **`Edible`**) and not a noun. This is because interfaces specify some common behaviour for objects of the classes that implement the interface. Classes that implement the **`Edible`** interface represent objects that can be eaten.

Methods in an interface are assumed to be either public or abstract. In order to implement an interface, a class needs to specify an **`implements`** clause in its class declaration and provide an implementation for every method in the interface. Just as a class **`extends`** a class (e.g. **`Lion extends Animal`**), a class **`implements`** an interface. The following is an example of a class that implements the **`Edible`** interface:

```
public class Apple implements Edible {
   public String howToEat() {
       return "Apple: Make apple juice";
   }
}
```

In the code above, the declaration for the *Apple* class specified implements *Edible* and the body of the class includes an implementation of the `howToEat()` method. When a class implements an interface, it implements all the methods defined in the interface with the exact signature and return type. A class can also implement more than one interface.

Just like a class, an interface is a data type and you can, therefore, use an interface as the type for a variable, parameter, or method return value. For example:

```
Edible food = getFood();
food.howToEat();
```

In the code above, the `getFood()` method returns an object that implements the `Edible` interface. The object is assigned to a variable called `food` of type `Edible`. In the second statement, the object's `howToEat()` method is called. You can also call the constructor of the class that implements the interface. For example:

```
Edible food = new Apple();
```

This was a brief introduction to interfaces. For more information, you can have a look at the **Oracle interfaces tutorial**.

STATIC METHODS

One of the ways we can specify a method is to make it static or not. A **static method** is a method that can be called without the instance or object of the class that it's in being called. First, let's look at what happens if we don't have a static method:

First, we have our `NameMaker` class with a method that concatenates a first name and a surname:

NameMaker.java

```
public class NameMaker {
    public String concat(String string1, String string2 ) {
        return string1 + " " + string2;
    }
}
```

Now we call that method to concatenate names we define in the Main method:

Main.java

```
public class Main {
   public static void main(String[] args) {

      String firstName = "Jon";
      String surName = "Snow";

      NameMaker name = new NameMaker();
      String fullName = name.concat(firstName, surName);

      System.out.println(fullName);
   }
}
```

Output:

```
Jon Snow
```

Do you see how we first have to create an instance of NameMaker (variable *name*) before we can call the `concat` method? Now let's see what happens if we make the `concat` method static:

NameMaker.java

```
public class NameMaker {
    public static String concat(String string1, String string2 ) {
        return string1 + " " + string2;
    }
}
```

Main.java

```
public class Main {
   public static void main(String[] args) {
   String firstName = "Jon";
   String surName = "Snow";
   String fullName = NameMaker.concat(firstName, surName);

   System.out.println(fullName);
   }
}
```

Output:

```
Jon Snow
```

See how we could use the `concat` method without a `NameMaker` object? That's because the memory of a static method is fixed in the ram, and so can be called without instantiation.

Compulsory Task 1

Follow these steps:

- Create a **Person.java** file that contains a Person class.
- Use a constructor to give the Person class the following attributes:
    - Height (integer)
    - Hair colour (string)
    - 18 or older (boolean)
- Extend the Person class to create Scientist and Doctor classes.
- The Scientist class also has the following features:
    - Lab coat colour (string)
    - Type of Scientist
    - Add a method in this class to use the lab coat colour to determine what kind of scientist someone is. The colours work as follows:
        - Green: Biologist
        - Purple: Physicist
        - White: Pharmacist
        - Blue: Climatologist
        - Yellow: Geneticist
        - Red: Zoologist.
    - Include a method that will print out a description of a Scientist object.
- The Doctor has the following features:
    - Ranking (string)
    - Years experience (integer)
    - The Doctor's ranking is determined based on their years of experience. Create a method to determine their ranking using the following information:
        - Medical student: 0-6 years
        - Intern: 7-8 years
        - Resident: 9-16 years
        - Fellow: 17-20 years
        - Attending 21+ years
    - Include a method that will print out a description of a Scientist object.

- Compile, save and run your file.

If you are having any difficulties, please feel free to contact our specialist team **on Discord** for support.

Rate us
Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.