Java

Abstract and Interfaces

Tobias Hanf, Manik Khurana

22. November 2021

Java-Course

Overview

Abstract

Abstract Class

The keyword **abstract** denotes an abstract class.

```
        public abstract class AbstractExample {

        }

```

- You can not create objects from an abstract class.
- Abstract classes can extend other abstract classes and can implement interfaces [1].
- Abstract classes can be extended by normal and abstract classes.

[1]Interfaces will be discussed later

Methods

An abstract class may has concrete methods and may has abstract methods. It can also has fields (attributes).

```java
        public abstract class AbstractExample {

            public void printHello() {
                System.out.println("Hello");
            }

            public abstract String getName();
        }

```

An abstract method forces the class to be abstract as well.

Subclasses

The subclass has to implement abstract methods or has to be abstract as well. All concrete methods will be regular inherited.

```java
        public class Example extends AbstractExample {

            @Override
            public String getName() {
                return "Example";
            }
        }

```

Polymorphism

Types

Definition: providing a single interface to entities of different types [1]

Two different types of polymorphism in Java:

- function/operator overloading
- subtyping (polymorphism)

[1] `https://www.stroustrup.com/glossary.html#Gpolymorphism`

Function overloading

Having the same function name but different parameters (types or number)

```
        public class Calc {

            double Add(double a, double b){
                System.out.println("Adding doubles");
                return a + b;
            }

            int Add(int a, int b){
                System.out.println("Adding integers");
                return a + b;
            }
        }

```

Function overloading

Calling the function with different types.

```
public class Main {

    public static main(String[] args){
        Calc calc = new Calc();

        calc.Add(3.5, 4.6);
        // prints: Adding doubles

        calc.Add(3, 4);
        // prints: Adding integers
    }
}

```

The compiler will check the types of the parameters an select the right function based on the signature.

Polymorphism

```java
public class Person {
    ...

    public void sendEmail(String message){
        System.out.println("To: " this.email
        + " \n" + message);
    }
}

public class Student extends Person {
    @Override
    public void sendEmail(String message){
        System.out.println("To: " this.email +
         " \n" + "Dear student :" + message);
    }
}

```

Polymorphism

Using the interface of the super class

```
        public class Main {
            public static main(String[] args){
                Person p1 = new Person("Bob");
                Person p2 = new Student("Alice");

                p1.sendEmail("Hello");
                // prints:  To: bob@school.org
                //          Hello

                p2.sendEmail("Hello");
                // prints:  To: alice@school.org
                //          Dear student: Hello
            }
        }

```

A object of type `Student` can be assigned to a variable of type `Person` because `Student` is a subclass of `Person`.

Static

Static Keyword

An object is an instance of a class with its attributes and methods.
The object is the actor and the class just a blueprint.

Static class members are not linked to a certain instance of the class.
Therefore the class can also be an actor.

Static class members are:

- static attributes, often called class variables
- static methods, often called class methods

Class Variables

In the setter `count` is addressed via `Example.count`. Using `this.count` is misleading, because `count` is a class variable.

```
        public class Example {

            public static count;

            public setCount(int count) {
                Example.count = count;
            }
        }

```

Class Variables - Test

The test prints the class variable `Example.count` which is altered by the different instances of the class *Example*.

```java
public class ExampleTest {

    public static void main (String[] args) {
        Example e1 = new Example();
        Example e2 = new Example();

        e1.setCount(4);
        System.out.println(Example.count); // prints: 4
        e2.setCount(8);
        System.out.println(Example.count); // prints: 8
    }
}

```

Class Methods

Static methods can be called without an object. They can modify class variables but not attributes (object variables).

```java
public class Example {

    public static count;

    public static void setCount(int count) {
        Example.count = count;
    }
}

```

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

    Example.setCount(4);
}

```

Static is an One-Way

Methods from objects can:

- access attributes (object variables)
- access class variables
- call methods
- call static methods

Class methods can:

- access class variables
- call static methods

Interfaces

An **interface** is a well defined set of constants and methods a class have to **implement**.

You can access objects through their interfaces. So you can work with different kinds of objects easily.

For Example: A post office offers to ship letters, postcards and packages. With an interface *Trackable* you can collect the positions unified. It is not important how a letter calculates its position. It is important that the letter communicate its position through the methods from the interface.

Interface Trackable

An interface contains method signatures. A signature is the definition of a method without the implementation.

```java
public interface Trackable {

    public int getStatus(int identifier);

    public Position getPosition(int identifier);
}

```

Note: The name of an interface often ends with the suffix -*able*.

Letter implements Trackable

```
public class Letter implements Trackable {

    public Position position;
    private int identifier;

    public int getStatus(int identifier) {
        return this.identifier;
    }

    public Position getPosition(int identifier) {
        return this.position;
    }
}

```

The classes *Postcard* and *Package* also implement the interface *Trackable*.

Access through an Interface

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

            Trackable letter_1 = new Letter();
            Trackable letter_2 = new Letter();
            Trackable postcard_1 = new Postcard();
            Trackable package_1 = new Package();

            letter_1.getPosition(2345);
            postcard_1.getStatus(1234);
        }

```

Two Interfaces

A class can implement multiple interfaces.

```java
public interface Buyable {

    // constant
    public float tax = 1.19f;

    public float getPrice();
}

```

```java
public interface Trackable {

    public int getStatus(int identifier);

    public Position getPosition(int identifier);
}

```

Postcard implements Buyable and Trackable

```
public class Postcard implements Buyable, Trackable {

    public Position position;
    private int identifier;
    private float priceWithoutVAT;

    public float getPrice() {
        return priceWithoutVAT * tax;
    }

    public int getStatus(int identifier) {
        return this.identifier;
    }

    public Position getPosition(int identifier) {
        return this.position;
    }
}

```

Access multiple Interfaces

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

            Trackable postcard_T = new Postcard();
            Postcard postcard_P = new Postcard();
            Buyable postcard_B = new Postcard();

            postcard_T.getStatus(1234);
            postcard_B.getPrice();
            postcard_P.getStatus(1234);
            postcard_P.getPrice();
        }

```

`postcard_P` can access both interfaces.
`postcard_T` can access Trackable.
`postcard_B` can access Buyable.