Java

Abstract

Hannes Ueck, Jakob Krude

10. Dezember 2020

Java-Kurs

Overview

1. Visibility
    - Public and Private
    - Getter und Setter
    - Package
2. Interface
    - Overview
    - Example
    - Multiple Interfaces
3. Abstract

Visibility

Public and Private

Until now we used **public** as standard visibility for methods and attributes. The alternative is **private**.

A public attribute is accessible from all classes.

A private attribute is only accessible from the own class.

Visibility dictates Accessibility.

Example - public

```
1  public class Test {
2
3      public int number;
4  }
5
```

```
1  public static void main (String[] args) {
2
3      Test test = new Test();
4      test.number = 16; // write
5      System.out.println(test.number); // read
6  }
7
```

Example - private

```
1     public class Test {
2
3         private int number;
4     }
5
```

```
1     public static void main (String[] args) {
2
3         Test test = new Test();
4         // access to test.number is impossible
5     }
6
```

Example - private

A public/private method has access to a private attribute of the same class.

```
1     public class Test {
2
3         private int number;
4
5         public void changeNumber() {
6             this.number = 3;
7             // access is possible
8         }
9     }
10
```

Access Attributes through Methods

Methods like `getAttributeX()` are called **getter**.

```
1     public class Test {
2
3         private int number;
4
5         public int getNumber() {
6             return this.number;
7         }
8     }
9
```

Modify Attributes through Methods

Methods like `setAttributeX(value)` are called **setter**.

```
1    public class Test {
2
3        private int number;
4
5        public void setNumber(int number) {
6            this.number = number;
7        }
8    }
9
```

Modify Attributes through Methods

A setter can check inputs and depending on the result modify attributes.

```
1   public class Test {
2
3       // number is always greater than zero
4       private int number;
5
6       public void setNumber(int number) {
7               if (number > 0) {
8               this.number = number;
9           }
10      }
11  }
12
```

Package

Packages help to organize large software with many classes.

Every **package** comes with a corresponding **namespace**. The package *myPackage* creates the namespace *myPackage*.

Inside a namespace you can not use the same name for two classes. But there can be a class *Test* in namespace myPackage and a class *Test* in the namespace *myOtherPackage*.

Example 1

The package declaration is above the class declaration.

```
1     package hello;
2
3     public class Number {
4
5         public int n1;
6
7         public void setN1(int n1) {
8                 this.n1 = n1;
9         }
10    }
11
```

Example 1 - Import

The class *Number* from the package *hello* will be imported.

```
1   package notHello;
2
3   import hello.Number;
4
5   public class Test {
6
7       public static void main (String[] args) {
8
9           Number number = new Number();
10          number.n1 = 16;
11      }
12  }
13
```

Protected

The attribute *n1* has now the visibility **protected**. This means *n1* it is visible for everyone in the same namespace *hello*.

```
1   package hello;
2
3   public class Number {
4
5       protected int n1;
6
7       public void setN1(int n1) {
8               this.n1 = n1;
9       }
10  }
11
```

Example 2

Number.n1 is protected, hence not visible for the class *Test*. The access to *n1* is possible through the public setter.

```
1   package notHello;
2
3   import hello.Number;
4
5   public class Test {
6
7       public static void main (String[] args) {
8
9           Number number = new Number();
10          number.setN1(16);
11      }
12  }
13
```

Interface

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.

```
1    public interface Trackable {
2
3        public int getStatus(int identifier);
4
5        public Position getPosition(int identifier);
6    }
7
```

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

Letter implements Trackable

```
1   public class Letter implements Trackable {
2
3       public Position position;
4       private int identifier;
5
6       public int getStatus(int identifier) {
7           return this.identifier;
8       }
9
10      public Position getPosition(int identifier) {
11          return this.position;
12      }
13  }
14
```

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

Access through an Interface

```
1     public static void main(String[] args) {
2
3         Trackable letter_1 = new Letter();
4         Trackable letter_2 = new Letter();
5         Trackable postcard_1 = new Postcard();
6         Trackable package_1 = new Package();
7
8         letter_1.getPosition(2345);
9         postcard_1.getStatus(1234);
10    }
11
```

Two Interfaces

A class can implement multiple interfaces.

```
1    public interface Buyable {
2
3        // constant
4        public float tax = 1.19f;
5
6        public float getPrice();
7    }
8
```

```
1    public interface Trackable {
2
3        public int getStatus(int identifier);
4
5        public Position getPosition(int identifier);
6    }
7
```

Postcard implements Buyable and Trackable

```
1   public class Postcard implements Buyable, Trackable {
2
3       public Position position;
4       private int identifier;
5       private float priceWithoutVAT;
6
7       public float getPrice() {
8           return priceWithoutVAT * tax;
9       }
10
11      public int getStatus(int identifier) {
12          return this.identifier;
13      }
14
15      public Position getPosition(int identifier) {
16          return this.position;
17      }
18  }
19
```

Access multiple Interfaces

```
1     public static void main(String[] args) {
2
3         Trackable postcard_T = new Postcard();
4         Postcard postcard_P = new Postcard();
5         Buyable postcard_B = new Postcard();
6
7         postcard_T.getStatus(1234);
8         postcard_B.getPrice();
9         postcard_P.getStatus(1234);
10        postcard_P.getPrice();
11    }
12
```

postcard_P can access both interfaces.

postcard_T can access Trackable.

postcard_B can access Buyable.

Abstract

Abstract Class

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

```
1    public abstract class AbstractExample {
2
3    }
4
```

Like an interface you can not create objects from an abstract class.
Abstract classes can extend other abstract classes and can implement interfaces.
Abstract classes can be extended by normal and abstract classes.

Methods

An abstract class may has concrete methods and may has abstract methods.

```
1    public abstract class AbstractExample {
2
3        public void printHello() {
4            System.out.println("Hello");
5        }
6
7        public abstract String getName();
8    }
9
```

An abstract method forces the class to be abstract as well.
Methods in an interface are also abstract but not denoted as such.

Subclasses

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

```
1     public class Example extends AbstractExample {
2
3         @Override
4         public String getName() {
5             return "Example";
6         }
7     }
8
```

Why using Abstract?

Abstract classes are used to minimize similar code in related classes.

Abstract classes can be used to offer a minimal implementation or default implementation for some methods.

Abstract class vs Interface

Abstract class

- abstract and non-abstract methods
- abstract class is extended by using keyword “extends”
- can implement multiple interfaces
- can have private, protected, public class members

Interface

- only abstract methods
- interface is implemented by using keyword “implements”
- can’t provide any implementation
- can only extend one other interface
- only public class members