Java

Interfaces & Polymorphie

Marcus Köhler

17. November 2018

Java-Kurs

Overview

1. Konstruktoren mit `super`
2. Weitere Kontrollstatements
   - switch
3. `static`
4. Interfaces
   - Überblick
   - Beispiel
   - Mehrere Interfaces
5. Polymorphie
   - Polymorphie mit Interfaces
   - Polymorphie mit Vererbung

Konstruktoren mit `super`

super()

Ein Konstruktor erzeugt eine neue Instanz einer Klasse.

Er ist sozusagen eine namenlose Methode mit dem return type <Klassenname>:

```
1  public class Test {
2      private int myInt;
3      private String myString;
4
5      public Test(int myInt) {
6          this.myInt = myInt;
7      }
8
9      public void setMyString(String myString) {
10         this.myString = myString;
11     }
12 }
13 [...]
14 Test t = new Test(5);
15 [...]
```

super()

Der `super()`-Aufruf ist nichts anderes als ein Aufruf des Konstruktors der Superklasse.
Dies ist nötig, da jede Subklasse automatisch eine Instanz ihrer Superklasse ist, welche vorher instanziert werden muss.

```
1 public class SubTest extends Test {
2     private char myChar;
3     // visible Attributes: myInt, myString, myChar
4     public SubTest(int myInt, char myChar) {
5         super(myInt); // calls the constructor with the myInt
      param of the SubTest constructor
6         this.myChar = myChar;
7         this.setMyString("My String");
8     }
9 }
```

Anders als ein normaler Aufruf des Konstruktors mittels `new` muss das `super()`-Statement keiner Referenz zugewiesen werden.

Weitere Kontrollstatements

if-else-if-else...

```
1  public static void main (String[] args) {
2      int address = 2;
3
4      if (address == 1) {
5          System.out.println("Dear Sir,");
6      } else if (address == 2) {
7          System.out.println("Dear Madam,");
8      } else if (address == 4) {
9          System.out.println("Dear Friend,");
10     } else {
11         System.out.println("Dear Sir/Madam,");
12     }
13 }
```

switch-case-default

```
1  public static void main (String[] args) {
2      int address = 2;
3
4      switch(address) {
5          case 1:
6              System.out.println("Dear Sir,");
7              break;
8          case 2:
9              System.out.println("Dear Madam,");
10             break;
11         case 4:
12             System.out.println("Dear Friend,");
13             break;
14         default:
15             System.out.println("Dear Sir/Madam,");
16             break;
17     }
18 }
```

switch-case-default

Um eine komplexe Verzweigung aufgrund einer Variable zu realisieren, kann man `switch` verwenden. Dies funktioniert mit allen primitiven Datentypen und `String`.
Die gegebene Variable wird mit dem Wert hinter dem Keyword `case` verglichen. Falls die Werte übereinstimmen, wird in den jeweiligen Block gesprungen. Falls keiner der Werte passt, wird der `default`-Block ausgeführt.

```
1  public static void main (String[] args) {
2      switch(intVariable) {
3          case 1:
4              doSomething();
5              break;
6          default:
7              doOtherThings();
8              break;
9      }
10 }
```

break

Nach dem letzten Statement des `case`-Blocks kann man mithilfe von `break` die Verzweigung verlassen.
Ohne `break` wird das Programm auch den folgenden Block ausführen, egal ob ein neuer `case`-Block beginnt.

```
1  public static void main (String[] args) {
2      switch( 1 ) {
3          case 1:
4              System.out.println("enter case 1"); //prints
5          case 2:
6              System.out.println("enter case 2"); //prints as well
7              break;
8          default:
9              System.out.println("enter default case"); //doesn't
       print
10             break;
11     }
12 }
```

return

Methoden mit dem return type `void` brauchen nicht zwingend ein `return`-Statement. Man kann jedoch mithilfe von `return` die Ausführung der Methode sofort beenden:

```
1 public void setNumber(int number) {
2     if(number < 0) return; // does not set the number if it's
      less than 0
3     this.number = number;
4 }
```

static

static

Ein Objekt ist eine Instanz einer Klasse, wobei die Klasse die Attribute und Methoden des Objekts definiert. Das Objekt ist sozusagen eine konkrete Umsetzung und die Klasse nur ein „Bauplan“.

static

Ein Objekt ist eine Instanz einer Klasse, wobei die Klasse die Attribute und Methoden des Objekts definiert. Das Objekt ist sozusagen eine konkrete Umsetzung und die Klasse nur ein „Bauplan“.

Statische Attribute und Methoden sind nicht and eine bestimmte Instanz(d.h. ein Objekt) der Klasse gebunden. Dadurch kann auch die Klasse an sich „aktiv“ sein.

`static`

Ein Objekt ist eine Instanz einer Klasse, wobei die Klasse die Attribute und Methoden des Objekts definiert. Das Objekt ist sozusagen eine konkrete Umsetzung und die Klasse nur ein „Bauplan“.

Statische Attribute und Methoden sind nicht and eine bestimmte Instanz(d.h. ein Objekt) der Klasse gebunden. Dadurch kann auch die Klasse an sich „aktiv“ sein.

Ein statischer Member(d.h. Klassen-Methoden und Klassen-Attribute) ist durch das Keyword `static` gekennzeichnet:

`static`

Ein Objekt ist eine Instanz einer Klasse, wobei die Klasse die Attribute und Methoden des Objekts definiert. Das Objekt ist sozusagen eine konkrete Umsetzung und die Klasse nur ein „Bauplan“.

Statische Attribute und Methoden sind nicht and eine bestimmte Instanz(d.h. ein Objekt) der Klasse gebunden. Dadurch kann auch die Klasse an sich „aktiv“ sein.

Ein statischer Member(d.h. Klassen-Methoden und Klassen-Attribute) ist durch das Keyword `static` gekennzeichnet:

static

Ein Objekt ist eine Instanz einer Klasse, wobei die Klasse die Attribute und Methoden des Objekts definiert. Das Objekt ist sozusagen eine konkrete Umsetzung und die Klasse nur ein „Bauplan“.

Statische Attribute und Methoden sind nicht and eine bestimmte Instanz(d.h. ein Objekt) der Klasse gebunden. Dadurch kann auch die Klasse an sich „aktiv“ sein.

Ein statischer Member(d.h. Klassen-Methoden und Klassen-Attribute) ist durch das Keyword `static` gekennzeichnet:

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

Klassen-Variablen

Im folgenden Setter wird auf `count` via `Example.count` zugegriffen. `this.count` ist in dem Fall falsch, da `count` der Klasse „gehört“.

```
1 public class Example {
2     public static count;
3
4     public setCount(int count) {
5         Example.count = count;
6     }
7 }
```

Klassen-Variablen

Der folgende Test gibt `Example.count` aus, welche durch die Instanzen von `Example` verändert wird:

```
1  public class ExampleTest {
2      public static void main (String[] args) {
3          Example e1 = new Example();
4          Example e2 = new Example();
5
6          e1.setCount(4);
7          System.out.println(Example.count); // prints: 4
8          e2.setCount(8);
9          System.out.println(Example.count); // prints: 8
10     }
11 }
```

Klassen-Methoden

Statische Methoden können ohne Objekt aufgerufen werden. Sie können Klassen-Variablen, aber nicht die Attribute von Objekten verändern.

```
1  public class Example {
2      public static count;
3      private String objAttr;
4
5      public static void setCount(int count) {
6          Example.count = count;
7      }
8
9      public static void setObjAttr(String attr) {
10         objAttr = attr; //this will produce an error at compile-
       time
11     }
12 }
```

```
1  public static void main (String[] args) {
2      Example.setCount(4);
3  }
```

`static` geht nur in eine Richtung

Objektmethoden können:

- auf Attribute zugreifen

Klassen-Methoden können:

`static` geht nur in eine Richtung

Objektmethoden können:

- auf Attribute zugreifen
- auf Klassen-Variablen zugreifen

Klassen-Methoden können:

`static` geht nur in eine Richtung

Objektmethoden können:

- auf Attribute zugreifen
- auf Klassen-Variablen zugreifen
- Methoden aufrufen

Klassen-Methoden können:

`static` geht nur in eine Richtung

Objektmethoden können:

- auf Attribute zugreifen
- auf Klassen-Variablen zugreifen
- Methoden aufrufen
- statische Methoden aufrufen

Klassen-Methoden können:

`static` geht nur in eine Richtung

Objektmethoden können:

- auf Attribute zugreifen
- auf Klassen-Variablen zugreifen
- Methoden aufrufen
- statische Methoden aufrufen

Klassen-Methoden können:

- auf Klassen-Variablen zugreifen

`static` geht nur in eine Richtung

Objektmethoden können:

- auf Attribute zugreifen
- auf Klassen-Variablen zugreifen
- Methoden aufrufen
- statische Methoden aufrufen

Klassen-Methoden können:

- auf Klassen-Variablen zugreifen
- statische Methoden aufrufen

Was tun, wenn...?

Was tun, wenn...

- ...man einige Klassen hat, die gleiches Verhalten umsetzen?

Was tun, wenn...?

Was tun, wenn...

- ...man einige Klassen hat, die gleiches Verhalten umsetzen?

  Die gemeinsamen Methoden in eine Superklasse auslagern!

Was tun, wenn...?

Was tun, wenn...

- ...man einige Klassen hat, die gleiches Verhalten umsetzen?

  Die gemeinsamen Methoden in eine Superklasse auslagern!
- ...man einige Klassen hat, die die gleichen Methoden umsetzen sollen, wobei die Umsetzung von der spezifischen Klasse abhängt?

Interfaces

Interfaces

Ein `interface` ist eine fest definierte Vorschrift, die von Klassen *implementiert* werden müssen. Interfaces können die folgenden Dinge definieren:

- Methodensignaturen

Interfaces

Ein `interface` ist eine fest definierte Vorschrift, die von Klassen *implementiert* werden müssen. Interfaces können die folgenden Dinge definieren:

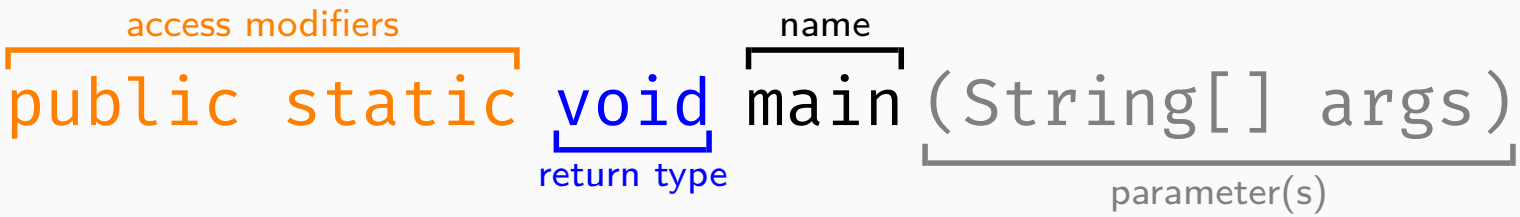
- Methodensignaturen
- Klassen-Konstanten(Variablen mit den modifiers `static` und `final`)

Interfaces

Ein `interface` ist eine fest definierte Vorschrift, die von Klassen *implementiert* werden müssen. Interfaces können die folgenden Dinge definieren:

- Methodensignaturen
- Klassen-Konstanten(Variablen mit den modifiers `static` und `final`)
- `default`-Methoden(seit Java 8)

Recap: Funktionssignaturen



`interface` Trackable

Ein simples Interface für einen Tracking-Service könnte folgendermaßen aussehen:

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

Viele Interfaces enden mit dem Suffix `-able`.

`interface` Trackable

Ein simples Interface für einen Tracking-Service könnte folgendermaßen aussehen:

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

Viele Interfaces enden mit dem Suffix `-able`.
Alle Klassen, die dieses Interface implementieren, müssen diese beiden Methoden umsetzen.

Letter implements Trackable

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

Die Klassen `Postcard` und `Package` implementieren auch `Trackable`.

Zwei Interfaces

Eine Klasse kann zwei Interfaces implementieren:

```java
public interface Buyable {
    // constant
    public float tax = 1.19f; // automatically gets changed to
    public static final float ...

    public float getPrice();
}
```

```java
public interface Trackable {
    public int getStatus(int identifier);

    public Position getPosition(int identifier);
}
```

Postcard implements Buyable, 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 }
```

Polymorphie

Zugriff auf Interfaces

Man kann von einem Objekt verlangen, dass sie ein bestimmtes Interface umsetzt.
Hierzu wird der Name des Interfaces bei der Deklaration wie ein Objekt verwendet:

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

Auf mehrere Interfaces zugreifen

Falls eine Klasse mehrere Interfaces implementiert, kann man sich „aussuchen“, welches man verlangt:

```
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 }
```

`postcard_P` kann beide Interfaces verwenden.
`postcard_T` kann `Trackable` verwenden.
`postcard_B` kann `Buyable` verwenden.

Polymorphie mit Vererbung

Ähnlich wie bei Interfaces kann man auch von Objekten verlangen, dass sie Unterklassen einer bestimmten Klasse sind.
Dazu kann man einer Referenz der Superklasse einfach eine Instanz einer passenden Unterklasse übergeben:

```
1  [...]
2      public static void main(String[] args) {
3          Delivery letter = new Letter();
4          Delivery package = new Package();
5
6          letter.setAdress("Faculty of CS, Dresden");
7          package.setAdress("cafe ASCII, Dresden");
8
9          package.printAdress(); //prints "Faculty of CS, Dresden"
10         letter.printAdress(); //prints "a letter for cafe ASCII,
       Dresden"
11     }
12 [...]
```

Limitierungen von Polymorphie

Wenn man mit Polymorphie arbeitet, muss man auf einige Besonderheiten achten:

- Man kann nur auf die Methoden zugreifen, die auch in dem jeweiligen Interface bzw. Superklasse definiert sind

Limitierungen von Polymorphie

Wenn man mit Polymorphie arbeitet, muss man auf einige Besonderheiten achten:

- Man kann nur auf die Methoden zugreifen, die auch in dem jeweiligen Interface bzw. Superklasse definiert sind
- Wenn man Polymorphie mit Vererbung verwendet, sollte man(falls nötig) die jeweilige Methode überschreiben (`@Override`)