Objektorientierte Programmierung - Skript

... mit Java

1 Klassen & Objekte

Seit Beginn der 90er Jahre hat sich die objektorientierte Programmierung (OOP) in der Praxis mehr und mehr durchgesetzt.

Wenn du ein Computerprogramm in einer objektorientierten Sprache schreibst, dann erstellst du in deinem Computer ein **Modell eines Ausschnitts der realen Welt**. Dieser Ausschnitt setzt sich aus den Objekten zusammen, die im Anwendungsbereich vorkommen.

Man spricht dann von der objektorientierten Modellierung (Modell = Abbild der Realität).
Die Objekte des Anwendungsbereichs unterscheiden sich je nach Programm, das du schreibst. Es können Nutzer und Nachrichten sein, wenn du an einem sozialen Netzwerk arbeitest, oder Monster, wenn du ein Computerspiel schreibst.

1.1 Klasse

Eine Klasse ist eine Schablone zur Beschreibung einer Menge von Objekten mit gemeinsamer Struktur und gemeinsamen Verhalten sowie Erzeugung solcher Objekte.

1.2 Objekt

Ein Objekt wird durch Zustand, Verhalten und Identität charakterisiert. Struktur und Verhalten ähnlicher Objekte sind durch ihre gemeinsame Klasse definiert.
Ein Objekt ist ein Exemplar (Instanz) einer Klasse, das sich entsprechend der Definition der Klasse verhält.

1.3 Klassendiagramm

Eine Klasse kann übersichtlich in einem Klassendiagramm dargestellt werden.

2 Klassendefinition

Befassen wir uns mit dem Quelltext einer Klasse. Eine Klasse besteht aus drei Basiselementen:

- Instanzvariablen (Datenfelder, Attribute)
- Konstruktoren

- Methoden

Die Instanzvariablen speichern die Daten, die ein Objekt benutzt.

Die Aufgabe der Konstruktoren ist es sicherzustellen, dass ein neu erzeugtes Objekt in einen vernünftigen Anfangszustand gesetzt wird. Wir bedienen hierbei das Bild einer Hebamme.

Die Methoden implementieren das Verhalten eines Objekts; sie liefern seine Funktionalität. Methoden enthalten einfache Arbeitsanweisungen, aber auch bedingte Anweisungen (if, if-else,..).

Beispiel: Klasse Person

```
public class Person
{
   private String vorname;
   private String nachname;

   public Person(String vorname, String nachname)
   {
      this.vorname = vorname;
      this.nachname = nachname;
   }

   public String getVorname()
   {
      return this.vorname;
   }

   public void setVorname(String vorname)
   {
      this.vorname = vorname;
   }

   public String getNachname()
   {
      return this.nachname;
   }

   public void setNachname(String nachname)
   {
      this.nachname = nachname;
   }
}
```

2.1 Kopf der Klasse

Der Kopf der Klasse:

```
public class Klassenname
{
```

```
   Innenteil der Klasse
}
```

Im Innenteil einer Klasse werden die Datenfelder, Konstruktoren und Methoden definiert, d.h. das alles, was die Klasse beschreibt, innerhalb der zwei geschweiften Klammern stehen muss.

2.2 Instanzvariablen / Datenfelder / Attribute

Instanzvariablen speichern Daten dauerhaft in einem Objekt. Sie definieren die **Eigenschaften** einer Klasse.
Die Klasse Person hat zwei Instanzvariablen:

- **vorname** ist eine Instanzvariable zur Speicherung des Vornamens einer Person.
- **nachname** ist eine Instanzvariable zur Speicherung des Nachnamens einer Person.

Für Instanzvariablen gelten folgende Punkte:

- Sie beginnen normalerweise mit dem reservierten Wort **private**. Auf private Instanzvariablen kann nur innerhalb der Klasse zugegriffen werden.
- Sie enthalten einen **Typnamen** (wie **int** für Ganze Zahlen, **double** für Gleitkommazahlen, **String** für Zeichenketten, **boolean** für boolesche Werte also true oder false, Person für Datenobjekte der Klasse Person, usw.), auch Datentyp genannt.
- Sie enthalten einen vom Benutzer gewählten **Namen** für die Variable.
- Sie enden mit einem Semikolon **;**

2.3 Konstruktoren

Konstruktoren haben eine ganz besondere Aufgabe. Sie sind verantwortlich dafür, dass ein Objekt unmittelbar nach seiner Erzeugung verwendet werden kann.

In gewisser Hinsicht kann ein Konstruktor mit einer Hebamme verglichen werden: Er ist dafür verantwortlich, dass das Objekt ordentlich ins Leben gerufen wird. Sobald ein Objekt erzeugt worden ist, spielt der Konstruktor im Leben des Objekts keine weitere Rolle und kann auch nicht mehr aufgerufen werden.

Eines der besonderen Merkmale des Konstruktors ist, dass er genauso heißt wie die Klasse, in der er definiert ist – in unserem Beispiel **Person**. Der Name des Konstruktors folgt direkt, d.h. ohne etwas dazwischen, auf das Wort **public**.

```
public Person(String vorname, String nachname)
  {
     this.vorname = vorname;
     this.nachname = nachname;
  }
```

In unserem obigen Beispiel bekommen die Datenfelder **vorname** und **nachname** Werte zugewiesen, da wir diese noch kennen.

Ein wichtiger Punkt an dieser Stelle ist, dass der Vorname und der Name an einer Stelle **außerhalb** der Person festgelegt wird und dass diese Information in die Person „hineingegeben“ werden muss.

Eine Aufgabe des Konstruktors ist, diesen Wert entgegen zunehmen und ihn in der Instanzvariable name bzw. vorname der neu erzeugten Person abzulegen. Auf diese Weise kann die Person (Objekt) den Wert behalten, ohne dass du die Person (Objekt) immer wieder daran erinnern musst.

Parameter

Parameter sind Variablen, die im Kopf eines Konstruktors oder einer Methode definiert werden:

```
public Person(String vorname, String nachname)
{
   this.vorname = vorname;
   this.nachname = nachname;
}
```

Ein Parameter wird als eine Art **temporärer Bote** verwendet, der Daten außerhalb des Konstruktors bzw. der Methode entgegennimmt und in den Konstruktor bzw. in die Methode hineinreicht, sodass sie dort zur Verfügung stehen.

Hier übergibt beispielsweise der **Parameter vorname**, den entgegengenommen Wert an das **Datenfeld vorname** (**this** steht hierbei für **dieses** Objekt).

Objekt erzeugen bzw. Konstruktoraufruf

```
//Deklaration einer Variable und Zuweisung eines Objekts
Klassenname variableX = new Klassenname();
```

Standardkonstruktor / Default-Konstruktor

Wird beim Anlegen einer Klasse kein Konstruktor angegeben, erzeugt der Compiler automatisch einen Standard-Konstruktor, der ohne die Angabe von Parametern aufgerufen wird.

2.4 Methoden

Eine Methode ist ein Programmteil, der in eine Klassendefinition eingebettet wird und aufgerufen werden kann. Die Methode kann uneingeschränkt auf alle Daten ihres Objekts zugreifen.

Syntax der Methodendefinition

```
[Zugriffmodifikator] Typ Name ([Parameter])
{
     [Anweisunge(-en;]
```

```
} // Ende der Methode
```

Methoden beginnen immer mit einem Kleinbuchstaben, jedes weitere Wort mit einem Großbuchstaben (z.B. zeigeInfo()).

2.4.1 Methoden mit Rückgabewert

```
public String getVorname()
   {
      return this.vorname;
   }
```

Du kannst eine Methode mit einem Rückgabewert als eine Frage an ein Objekt vorstellen. Der **Rückgabewert** einer Methode ist dann die Antwort auf die Frage.
In diesem Fall (getVorname) lautet die Frage: „Wie heißt die Person mit Vorname?“ Die Antwort hält die Person im **Datenfeld vorname** fest. Den Wert dieses Datenfelds liefert die Methode.

Wichtig ist dabei die Rückgabeanweisung:

```
return this.vorname;
```

Sie sorgt dafür, dass ein Wert vom Typ String zurückgeliefert wird, ganz so, wie es im Kopf der Methode definiert ist.

2.4.2 Methoden mit Parameter

In gleicher Weise, wie wir uns Methoden mit Rückgabewerten als Bitten um Informationen (Fragen) vorstellen können, können verändernde Methoden als Bitten auffassen, dass das gerufene Objekt seinen Zustand verändern soll.
Die Grundform einer verändernden Methode ist eine Methode, die einen einzigen Parameter übernimmt, dessen Wert dazu dient, das, was in einem Datenfeld gespeichert ist, zu überschreiben.

```
public void setVorname(String pVorname)
   {
      this.vorname = pVorname;
   }
```

Der Kopf der Methode **setVorname** definiert als Ergebnistyp **void** sowie einen Parameter betrag vom Typ **String**.

- Ein **Ergebnistyp void** besagt, dass eine Methode keinen Wert an ihren Aufrufer zurückliefert. Dies ist ein grundlegender Unterschied zu alle anderen Ergebnistypen. Im Rumpf der Methode wird deutlich, dass keine Rückgabeanweisung angegeben ist.
- Im Rumpf der Methode setVorname steht eine einzelne Anweisung, die eine **Zuweisung** darstellt. Der Wert der rechts vom = steht wird dem Datenfeld(Variable) zugewiesen bzw. abgespeichert: `this.vorname = pVorname;`

2.4.3 Methoden ohne Rückgabewert und ohne Parameter

Methoden mit ohne Rückgabewert und ohne Parameter sehen beispielsweise so aus:

```
public void druckeVornameNachname()
{
   // Den Vornamen und Nachnamen einer Person in der Konsole ausgeben.
   System.out.println("Mein Name: " + vorname + nachname );

}
```

- Der Kopf besagt, dass die Methode den Ergebnistyp void hat und keine Parameter annimmt.
- Der Rumpf enthält ein Kommentar und eine Anweisung, die für eine Ausgabe in der Konsole sorgt.

Eine Anweisung wie

```
System.out.println("Hallo");
```

bewirkt, dass die in doppelten Anführungszeichen stehenden Zeichenkette ausgegeben wird. Die grundlegende Form eines Aufrufs von println ist

```
System.out.println(etwas-das-wir-ausgeben-wollen);
```

Methoden überladen

Eine Methode überladen bedeutet, dass es eine Methode mit gleichem Namen, aber unterschiedlicher Parameterliste, mehrfach gibt.

3 Datenkapselung

Das Kernelement der Datenkapselung besteht darin, die internen Informationen eines Objekts vor unberechtigtem Zugriff zu schützen. Hierbei wird kontrolliert, welche Daten von außen sichtbar und zugänglich sind, während andere Informationen verborgen bleiben.

Zugriffmodifikatoren

Java bietet unterschiedliche Zugriffsmodifikatoren für Instanzvariablen und Methoden. Sie ermöglichen mittels der Schlüsselwörter private, protected oder public die Sichtbarkeit einzuschränken.

Zugriffsmodifikator	Klasse	Unterklassen	Welt
public	ja	ja	ja

Zugriffsmodifikator	Klasse	Unterklassen	Welt
protected	ja	ja	nein
private	ja	nein	nein

Vorteile

- Sicherheit: Schützt Daten vor unberechtigtem Zugriff und Manipulation.
- Flexibilität: Änderungen an Datenstrukturen können auf die Klasse beschränkt werden, ohne dass dies Auswirkungen auf andere Teile des Programms hat.

4 Generalisierung & Spezialisierung (Vererbung)

Die Generalisierung stellt ein zentrales Konzept objektorientierter Modellierung dar. Sie setzt zwei Klassen so in Beziehung, dass eine Klasse eine Verallgemeinerung des anderen darstellt. In der folgenden Abbildung ist die Klasse Gast eine Verallgemeinerung von Partyneuling bzw. Stammgast. Oder - umgekehrt – Partyneuling ist ein spezieller Gast. Die Umkehrung der Generalisierung ist Spezialisierung.

Die allgemeine Klasse wird als **Oberklasse** einer spezialisierten **Unterklasse** bezeichnet. Jede Instanz einer Unterklasse ist auch immer indirekt eine Instanz der Oberklasse, das bedeutet, dass die Eigenschaften (Attribute, Operationen, Beziehungen, etc.) von der Oberklasse auf die Unterklasse übertragen werden. Dies wird als **Vererbung** bezeichnet.
Neben den Eigenschaften der Oberklasse kann die Unterklasse darüber hinaus zusätzliche definieren oder die der Oberklasse abändernd überschreiben. Hier diesem Beispiel steht einem Objekt der `Unterklasse Partyneuling` auch die Operation `feiere()` der `Oberklasse Gast` zur Verfügung.
Durch die Verwendung der Generalisierungsbeziehung entsteht eine **Hierarchie**.

In einem UML-Klassedidagramm wird die Generalisierungsbeziehung wird mit einem **Pfeil mit nicht ausgefüllter dreieckiger Pfeilspitze** dargestellt. Er zeigt von der spezialisierten Klasse hin zur allgemeinen und generalisierten Klasse (Pfeilspitze).

In Java wird die Vererbung durch das Schlüsselwort `extends` beschrieben:

```
public class KlasseA
{...}

public class KlasseB extends KlasseA
{...}
```

Lesbarkeit

Die Vererbung kann durch die Zugriffsmodifikatoren für Instanzvariablen und Methoden eingeschränkt

werden. Eine Unterklasse erbt dann alle Instanzvariablen und Methoden die **nicht** `private` sind. Ist eine Instanzvariable durch `protected` gekennzeichnet, dann kann die Unterklasse diese Eigenschaften sehen.

Verhalten von Konstruktoren in der Vererbung

Konstruktoren werden nicht vererbt. Bei Bedarf muss man in den Unterklassen neue Konstruktoren definieren.

Wenn ein Objekt einer Unterklasse erzeugt wird, ruft der Konstruktor der Unterklasse automatisch den Standard-Konstruktor der Oberklasse auf. In Java kann man den Konstruktor auch expliziz durch das Schlüsselwort `super()` aufrufen. Grundsätzlich darf keine Anweisung vor dem Aufruf des Konstruktors der Oberklasse stehen.

Das ist aber nur nötig, wenn wir dem Konstruktor der Oberklasse Parameter übergeben müssen. Schauen wir uns dazu ein Beispiel an. Angenommen die Klasse Tier hat folgende Gestalt:

```
public class Tier {
    public String tiername;

    public Tier(String p_name) {
        tiername = p_name;
    }
}
```

Dann kannst du diesen Konstruktor aus der Unterklasse Gorilla wie folgt aufrufen:

```
public class Gorilla extends Tier {
    public double gewicht;

    public Gorilla(String p_name, double p_gewicht) {
        super(p_name);
                gewicht = p_gewicht;
    }
}
```

5 Abstrakte Klassen

Nicht immer soll eine Klasse ausprogrammiert werden. Manchmal möchten wir in einer Oberklasse lediglich Methoden für die Unterklassen vorgeben, ohne zu wissen, wie diese konkret ausprogrammiert werden.

Beispiel
Wir haben eine abstrakte `Klasse Tier` (Schlüsselwort: abstract):

```
public abstract class Tier extends Actor
{
```

```
    private String name;

    // Keine anstrakte Methode
    public void setName(String pName)
    {
        this.name = pName;
    }

    // Abstrakte Methode
    public abstract void lautGeben();
}
```

Von dieser abstrakten `Klasse Tier` dürfen wir keine Objekte erstellen. Die **abstrakte** `Klasse Tier` enthält auch eine abstrakte Methode `lautGeben()`, die nur einen Prototypen (Methodenkopf) darstellt. Neben abstrakten Methoden können auch konkrete Methoden in einer abstrakten Klasse definiert werden.

Die Unterklassen dieser abstrakten Klassen erben die Vorgabe diese Methode zu definieren oder selber eine abstrakte Klasse zu sein.

```
public class Schwein extends Tier
{
    public void lautGeben()
    {
        Greenfoot.playSound("cow.wav");
    }
}
```

6 Assoziation

Eine Beziehung zwischen zwei oder mehreren Klassen wird als Assoziation bezeichnet. Die Assoziation beschreibt eine sehr enge Form der Beziehung zwischen zwei Klassen, die das gegenseitige Zugreifen auf Elemente der Klasse (Attribute und Operationen) **ermöglicht**. Eine Assoziation beschreibt die Beziehung aller Objekte der beteiligten Klassen. Sie wird auf Objektebene hergestellt und auf Klassenebene modelliert.

Beispielsweise besteht zwischen der Klasse Kunde und der Klasse Girokonto eine Beziehung. Denn ein Kunde kann ein Girokonto haben und der Kontoinhaber eines Girokontos ist ein Kunde. Beziehungen zwischen Klassen (Assoziationen) werden genauer beschrieben durch **Rollen**, **Multiplizitäten** und **Navigierbarkeit**.

Rollen

Rollennamen beschreiben die Bedeutung der Objekte der assoziierten Klasse näher.

Aus der Sicht des Girokontos ist der Kunde der Kontoinhaber. Aus der Sicht des Kunden ist das

Girokonto sein Konto. Die Rolle einer assoziierten Klasse kann ggf. Auswirkungen auf die Multiplizitäten und die Navigierbarkeit haben.

Multiplizitäten

Multiplizitäten geben an mit wie vielen Objekten der assoziierten Klasse ein Objekt verbunden werden kann (vgl. Kardinalitäten bei ERDs). Sie stellen folglich Mengenverhältnisse zwischen Klassen dar.

Beispiel

Jeder Kunde hat kein oder ein Girokonto. Jedes Girokonto hat genau einen Kontoinhaber (= Kunden).

Im oben beschriebenen Beispiel kann ein Kunde ein Girokonto haben. Es ist aber auch vorstellbar, dass der Kunde bei der Bank kein Girokonto hat, sondern nur ein Sparkonto. Deswegen spricht man hier von einer Kann-Assoziation (Untergrenze 0). Dagegen muss ein Girokonto einen Kunden als Kontoinhaber haben, weswegen hier von einer Muss-Assoziation (Untergrenze > 0) gesprochen wird.

Mögliche Multiplizitäten sind:

0 .. 1	kein oder ein assoziiertes Objekt	KANN-Assoziation
1	genau ein assoziiertes Objekt	MUSS-Assoziation
*	kein, ein oder beliebig viele assoziierte Objekte	KANN-Assoziation
1 .. *	ein oder beliebig viele assoziierte Objekte	MUSS-Assoziation
n .. m	von n bis m assoziierte Objekte	-

Weitere Beispiele

Jeder Kunde hat kein, ein oder beliebig viele Girokonten. Jedes Girokonto hat ein oder be-liebig viele Kontoinhaber (= Kunden).

Jeder Kunde hat kein, ein oder beliebig viele Girokonten. Jedes Girokonto hat 1 oder 4 Kontoinhaber (= Kunden).

Navigierbarkeit

Bei der Navigierbarkeit wird die Frage beantwortet, wer kann auf wen zugreifen? Dabei unterscheidet man zwischen **unidirektionalen** und **bidirektionalen** Assoziationen.

Unidirektionale Assoziation

Jedes Objekt der Klasse Girokonto kennt seinen Kontoinhaber (= Kunde). Allerdings kennt ein Objekt der Klasse Kunde nicht seine Girokonten. Das heißt, man kann über das Girokonto z.B. den Namen

des Kontos ausgeben, aber nicht über den Kun-den den Kontostand seines Girokontos. Die Navigierbarkeit ist also nur in eine Richtung möglich, nämlich von Kunde zu Girokonto (Pfeil), aber nicht von Girokonto zu Kunde („X“)

Bidirektionale Assoziation

Jedes Objekt der Klasse Kunde weiß, wel-che Girokonten ihm zugeordnet sind. Umgekehrt weiß aber auch jedes Girokonto welche Kontoinhaber (= Kunden) es hat. Die Navigierbarkeit ist in beide Richtun-gen möglich. Das heißt, über einen Kun-den kann der Kontostand seines Girokon-tos herausgefunden werden und über ein Girokonto kann der Name des Kunden ermittelt werden.

Neben der Pfeilspitze und dem „X“ kann auch „nichts“ geschrieben wer-den. Dies bedeutet, dass noch keine Beziehung festgelegt wurde.

Mögliche Beziehungen

Pfeilspitze	Zugriff auf die Klasse möglich.
„X“	Kein Zugriff auf die Klasse möglich.
(nichts)	Keine Beziehung festgelegt.

7 Algorithmus

7.1 Definition

Ein Algorithmus ist eine eindeutige Handlungsvorschrift zur Lösung eines Problems oder einer Klasse von Problemen.

7.2 Formale Definition

Ein Algorithmus ist ein Verfahren, das

- durch eine **endliche** Beschreibung
- in einer **präzisen** (d.h. genau festgelegten) Sprache
- unter Verwendung **effektiver** (d.h. tatsächlich ausführbarer) und **elementarer** (Verarbeitungs-) Schritte

festgelegt wird.

(nach Broy, M.: Informatik: Eine grundlegende Einführung, Band 1. Springer-Verlag, Berlin, 2. Auflage, 1998)

7.3 Programme und Algorithmen

- Ein Programm ist eine (von vielen möglichen) Darstellungen eines bestimmten Algorithmus als Text.
- Dieser Text ist in einer speziellen Sprache, die vom Rechner automatisch interpretiert werden kann.
- Eine solche Sprache heißt Programmiersprache.

7.4 Struktur von Algorithmen

- Algorithmen haben eine sehr einfache Struktur.
- Jeder Algorithmus lässt sich aus vier verschiedenen Typen von Strukturelementen zusammensetzen.
    - Elementare Verarbeitungsschritte
    - Sequenz
    - Bedingte Anweisung
    - Wiederholung

7.4.1 Elementare Verarbeitungsschritte

„Unteilbare“ atomare Aktionen, z.B.:

- Aufruf einer Methode eines Objekts: k2.fuellen(schwarz);
- Ausgabe eines Wertes: System.out.print(Ergebnis);
- Berechnung eines Terms: c = Wurzel(a*a+b*b);

7.4.2 Sequenzen

- Eine Sequenz ist eine Folge von elementaren Verarbeitungsschritten, die unbedingt in der angegebenen Reihenfolge abgearbeitet werden.
- Je nach Programmiersprache sind verschiedene (oft auch mehrere) Trennzeichen zwischen den Anweisungen einer Sequenz zugelassen, z.B.: Strichpunkt

```
move(1);
turn(90);
putLeaf();
```

7.4.3 Auswahlstruktur

- **einseitige** Auswahlstruktur: `if(Bedingung){Anweisung(en)}`
  Wenn für den Fall, dass die Bedingung FALSCH ist, keine Sequenz angegeben wird, spricht man von einer einseitigen Auswahlstruktur oder einer bedingten Anweisung.
- **zweiseitige** Auswahlstruktur: `if(Bedingung){Anweisung(en)}else{Anweisung(en)}`
- **mehrseitige** Auswahlstrukur: else-if oder switch(case)

7.4.4 Schleifen (Wiederholung)

- while-Schleife und do-while-Schleife

```
while (laufbedingung) {
  // anweisung(en);
}
```

```
do {
  // anweisung(en);
}
while (laufbedingung);

}
```

- for-schleife

```
for (int i = 0; i < 5; i++) {
  System.out.println(i);
}
```

[informatik, thema, java, oop]