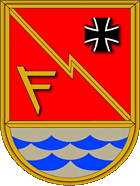
Schule Informationstechnik

der Bundeswehr



**Sprachausbildung Java**



**Übungen**

**"Objektorientierte Programmierung"**

Klassen, Objekte, Attribute und Methoden

# Verständnis

Wir beginnen zunächst mit einer Reihe von Verständnisfragen. Vervollständigen Sie folgende Aussagen bzw. geben Sie an, welche der unten stehenden Aussagen korrekt ist:

1. Um den kontrollierten Zugriff auf Datenelemente einer Klasse sicherzustellen, werden die Datenelemente (Attribute) normalerweise als private deklariert und durch \_\_\_\_\_\_\_\_\_\_\_ gelesen oder geändert.
2. Gegeben sei der folgende Ausschnitt einer Klassendefinition:
3. class Demo { double zahl; /\* … \*/}

Ist das Datenelement zahl dann als public deklariert?

1. Elemente von verschiedenen Klassen

* müssen unterschiedliche Namen haben.
* können den gleichen Namen haben.

1. Gegeben sei folgende Klassendefinition:
2. class Test
3. {
4. int iCount;
5. public void setCount(int iN){
6. } // iN -> iCount
7. }

Für ein Objekt myTest der Klasse Test ist folgende Anweisung (unter bestimmten Bedingungen) zulässig:

* myTest.iCount = 7;
* Test.iCount = 7;
* myTest.setCount(7);

1. Der Name eines Konstruktors ist identisch mit dem Namen \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.
2. Der Return-Typ eines Konstruktors …

* existiert nicht.
* ist mit dem Typ der Klasse identisch.
* ist void.

1. Konstruktoren derselben Klasse müssen unterscheidbar sein durch ihre \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

# UML Pizzabäckerei

Modellieren Sie eine Pizzabäckerei, zu der folgende Informationen vorliegen:

* Eine Pizzabäckerei hat mindestens einen Backofen und
* mindestens einen Pizzabäcker.
* Sie benötigt ein Lager, das u.a. die Zutaten für Pizzen enthält.

In der Bäckerei können sich Kunden aufhalten, die auf die bestellten Pizzen warten.

Erstellen Sie nun ein Klassenmodell (UML Klassendiagramm) mit den wesentlichen Klassen und beschreiben Sie jeweils die Art der Beziehung, die Sie zwischen diesen Klassen ermitteln.

# Sparschwein

Implementieren Sie eine Klasse PiggyBank zur Verwaltung der Münzen in einem Sparschwein. Die Klasse besitzt folgende Attribute:

* Zähler für vier Arten von Geldstücken (Anzahl-1-Cent, -10-Cent, -50-Cent und -1-Euro-Stücke)
* maximale Anzahl Geldstücke, die in das Sparschwein passen
* Flag, um anzuzeigen, dass das Sparschwein aufgebrochen wurde

Darüber hinaus deklariert die Klasse folgende Methoden:

* init() Initialisiert jedes Attribut zur Darstellung eines leeren   
   Sparschweins. Die Methode erwartet als Argument die maximale   
   Anzahl von Münzen, die in das Sparschwein passen.
* add1Cents() "wirft" eine übergebene Anzahl von 1-Cent-Münzen in das   
   Sparschwein und liefert den Return-Wert 0, falls alle Münzen in   
   das Sparschwein passen. Wann das Sparschwein "überläuft", liefert   
   die Methode die Anzahl Münzen, die nicht mehr hinein passen.   
   Sollte das Sparschwein aufgebrochen worden sein, können keine   
   Münzen eingeworfen werden. Bei negativer Anzahl an Münzen,   
   wird der negative Wert zurückgeliefert.
* add10Cents() / add50Cents() / add1Euros()  
   analog zur Methode add1Cents() für den jeweiligen Münz-Typ
* isEmpty() liefert true, falls das Sparschwein leer ist, sonst false.
* isFull() liefert true, falls das Sparschwein voll ist, sonst false.
* isBroken() liefert true, falls das Sparschwein aufgebrochen ist, sonst false.
* breakPiggy() bricht das Sparschwein auf. Der Zähler der Geldstücke wird auf 0   
   zurückgesetzt. Der Return-Wert ist der angesparte Geldbetrag in   
   Cents.

# Sparschwein testen

Testen Sie die Klasse PiggyBank mit einem Programm in einer separaten Quelldatei PiggyBankTest. Legen Sie ein Objekt der Klasse an, welches bis zu 500 Münzen speichern kann. "Werfen" Sie verschiedene Münzen in das Sparschwein, bis es voll ist. Brechen Sie dann das Sparschwein auf und zeigen Sie den gesparten Betrag an.

# Personalarray

Implementieren Sie eine Klasse “Mitarbeiter“. Mit den Attributen Vorname, Name, Personalnummer, Posten, Eintrittsdatum und Gehalt. Implementieren Sie entsprechende Konstruktoren und weitere notwendige Methoden.

Schreiben Sie ein Programm mit einem Array, in welchem 50 Mitarbeiter gespeichert werden können. Implementieren Sie Methoden, mit denen Mitarbeiter aufgrund der Personalnummer oder des Nachnamens aus dem Array ausgewählt und ausgegeben werden können.

Beispielinteraktion:

Personalnummer:

*1234*

Gefundene Einträge:

1. Max Mustermann, Portier, Eingestellt am 2011-04-01, Gehalt: 1500.00€

# Lehrgangsobjekte

Implementieren Sie eine Klasse “Lehrgang“. Überlegen Sie sich, welche Attribute (außer Nummer, Name, Dauer) für diese Klasse sinnvoll sind.

Erzeugen Sie, in einem Programm, zwei Objekte dieser Klasse, die zwei der Lehrgänge repräsentieren, die Sie besuchen bzw. in der letzten Zeit besucht haben. Ihr Programm soll die Bezeichnung und die Nummer der Lehrgänge ausgeben.

# Topfspiel

Wir haben nun die Aufgabe ein kleines Topfspiel, mit objektorientiertem Ansatz, zu programmieren. Es ist ein Konsolenspiel zu realisieren, in dem die beiden Spieler an einem Rechner spielen. Die Eingaben erfolgen über Tastatur und die Ausgaben werden im Konsolenfenster dargestellt. Folgende Anforderungen soll das Spiel erfüllen:

* Das Spiel wird mit zwei Spielern gespielt.
* Jedem Spieler gehört ein kleiner, am Anfang leerer Topf.
* Nach Aufforderung des aktuellen Spielers (Spieler 1 oder Spieler 2) führt dieser einen möglichen Spielzug aus.
* Es wird abwechselt ein Spielzug von den Spielern durchgeführt. Spieler 1 beginnt das Spiel.
* Folgende 3 Aktionen sind möglich:
  + 1 – 3 Münzen in den eigenen Topf ablegen
  + Eine Münze aus dem gegnerischen Topf herausholen
  + Tauschen der Töpfe.
* Die möglichen Aktionen sind vor jedem Spielzug dem aktuellen Spieler anzuzeigen.
* Die ausgewählte Aktion gibt der Benutzer über die Tastatur dem Programm bekannt.
* Wenn die Aktion: Topftauschen ausgewählt wurde, dann ist diese Aktion für zwei weitere Züge nicht mehr erlaubt. Es ist eine Sperre zu integrieren.
* Nach einem Spielzug ist der aktuelle Spielstand auszugeben.
* Fehlerhafte Eingaben sind abzufangen und mit einer Meldung zu bestätigen.
* Gewonnen hat der Spieler, der zuerst 7 Münzen in seinem Topf hat.
* Eine Gewinnermeldung ist darzustellen und das Programm beendet sich.

1. Modellieren Sie die zwei Klassen:

* Eine Topf-Klasse: Topf, die die Anzahl der Münzen im Topf verwaltet und Methoden für das Hineinlegen oder Entfernen von Münzen anbietet, den aktuellen Topfinhalt ausgibt sowie Auskunft gibt, ob der Topf voll ist.
* Eine Spiel-Klasse: TopfSpiel, die den aktuellen Spieler, die Töpfe der Spieler und die Spielsperre verwaltet. Die benötigten Methoden entnehmen Sie dem vorgegebenen Hauptprogramm:

1. package topfSpiel;
2. public class TopfSpiel {
3. public static void main( String[] args ) {
4. TopfSpiel spiel = new TopfSpiel();
5. while ( !spiel.istBeendet() ) {
6. spiel.runde();
7. }
8. System.out.println( "Sieger ist Spieler " +   
    spiel.getSieger() );
9. }
10. }
11. Implementieren Sie das Spiel.

Vererbung und Polymorphie

# Verständnisfragen

Zunächst wieder ein paar Verständnisfragen vorne weg:

1. Eine "Hat-Beziehung" entsteht zwischen zwei verschiedenen Klassen, wenn

* eine Klasse von der anderen Klasse abgeleitet ist
* ein Attribut einer Klasse vom Typ der anderen Klasse ist
* eine Klasse innerhalb der anderen Klasse definiert ist

1. Eine in einer abgeleiteten Klasse überschriebene Methode kann die entsprechende public-Methode der Basisklasse aufrufen, und zwar mit Hilfe des Operators \_\_\_\_\_.
2. Welche Beziehung besteht jeweils zwischen den Klassen
3. Person – Pkw
4. Soldat – PK
5. KpChef – KpFw
6. Kp – KpFw
7. Firma – Angestellter
8. Firma – Kunde
9. Tier – Affe
10. Zoo – Affenhaus
11. Was ist ein Konstruktor?
12. Was sind die wesentlichen Konzepte der objektorientierten Programmierung?

# Dreidimensionale Punkte

Die folgende Java-Klasse modelliert zweidimensionale Punkte:

1. public class Punkt2D {
2. // Koordinaten
3. public int xKoord;
4. Public int yKoord;
5. public Punkt2D (int xKoord, int yKoord) {
6. this.xKoord = xKoord;
7. this.yKoord = yKoord;
8. }
9. @Override
10. public boolean equals (Object obj) {
11. boolean erg=false;
12. if(obj instanceof Punkt2D){
13. Punkt2D other = (Punkt2D) obj;
14. erg = (other.xKoord == this.xKoord)   
     && (other.yKoord == yKoord);
15. }
16. return erg;
17. }
18. public static void main(String[] args){
19. Punkt2D p1 = new Punkt2D(2,2);
20. Punkt2D p2 = new Punkt2D(3,2);
21. System.out.println(p1.equals(p2));
22. }
23. }

Definieren Sie im gleichen Stil zur Klasse Punkt2D eine Unterklasse Punkt3D, deren Objekte dreidimensionale Punkte modellieren und daher zusätzlich eine z-Koordinate enthalten. Die Klasse Punkt3D soll folgende Elemente bereitstellen. Wo immer es geht, sollen sich diese Elemente auf Elemente der Oberklasse oder auf andere Elemente der eigenen Klasse abstützen:

1. Einen Konstruktor mit drei int-Parametern, mit denen die Koordinaten belegt werden.
2. Einen parameterlosen Konstruktor, der ein Exemplar des Ursprungspunkts (0,0,0) erzeugt.
3. Eine equals-Methode zum Vergleich von Punkt3D-Objekten mit anderen Punkt3D-Objekten.
4. Ergänzen Sie die beiden Klassen um eine sinnvolle toString() Methode.

# **Schulverwaltungssoftware**

In der Schulverwaltungssoftware werden Menschen verwaltet, diese zeichnen sich durch ihren Namen und ihr Alter aus. Es gibt zwei spezielle Arten von Menschen an der Schule, Lehrer und Schüler. Lehrer zeichnen sich zusätzlich durch ein Unterrichtsfach aus.

Bei den Schülern sind andere Informationen interessant. Jeder Schüler hat eine Klassenstufe (5 bis 12). Außerdem bezahlen die Schüler einen jährlichen Beitrag in die Schulkasse. Zur Berechnung dieses Beitrags werden das Alter, die Klassenstufe und der Jahresbeitragsfaktor herangezogen. Der Jahresbeitragsfaktor liegt derzeit für alle Schüler bei 5.

Die Formel zur Berechnung des jährlichen Beitrags lautet:

Besondere Schüler sind die Klassensprecher, die aufgrund ihrer großartigen Verdienste einen niedrigeren, individuellen Jahresbeitragsfaktor haben (3 oder 4).

1. Erstellen Sie ein Klassendiagramm.
2. Programmieren Sie die Klassen, instanziieren Sie von allen Klassen ein Objekt und testen Sie die Funktionsfähigkeit.

# Fußball

Implementieren Sie die Klassen Spieler, Torwart, Angreifer und Verteidiger aus dem folgenden Diagramm.

Dabei wird die Methode training() beim Torwart und beim Verteidiger so überschrieben, dass ein anderer Text als beim generellen Spieler ausgegeben wird. In der Klasse Mannschaft ist ein Attribut auf den Torwart zu setzen; zudem sind zwei Arrays mit 4 bzw. 6 Elementen auf die verteidigung bzw. auf den angriff zu erstellen. Beim training() des Spielers soll "Jogging" auf der Konsole ausgegeben werden, beim Verteidiger „Kniebeugen" und beim Torwart „Hochsprung".

Zusatzaufgabe:

Implementieren Sie in der Klasse Mannschaft zusätzlich eine Methode Namens training(), welche alle Spieler trainieren lässt. Hierzu verwenden Sie einfach die Delegation auf die Klassen Torwart, Verteidigung und Angreifer.

# Rennschnecke

1. Erstellen Sie eine Klasse "Rennschnecke".
2. Rennschnecken sollen folgende Eigenschaften (Attribute) besitzen:
   * einen Namen,
   * eine Rasse,
   * eine Maximalgeschwindigkeit (Veränderung der Strecke pro Schritt),
   * die Schnecke soll wissen welchen Weg sie bereits zurückgelegt hat.
3. Erstellen Sie für die Klasse Rennschnecke einen Konstruktor, der den Instanzvariablen beim Erstellen einer neuen Instanz (Objekt zu dieser Klasse) Werte zuweist.
4. Legen Sie in der Klasse "Rennschnecke" eine Methode "krieche()" an, welche die Schnecke abhängig von ihrer Maximalgeschwindigkeit eine zufällige Strecke weiter bewegt. Soll heißen: Sie kriecht eine zufällige Strecke größer null und kleiner ihrer Maximalgeschwindigkeit. Nimm als Zeiteinheit 1 an.
   * Tipp: Schauen Sie sich die Methode Math.random() aus der Java API an.
5. Legen Sie in der Klasse "Rennschnecke" eine Methode "public String toString()" an, welche die Daten der Schnecke als String zurückgibt.
6. Testen Sie Ihre Klasse, indem Sie probehalber ein Rennschneckenobjekt erzeugen und seine Daten auf der Konsole ausgeben.
   * Tipp: Verwenden Sie zum Ausgeben der Daten die toString() Methode der Rennschnecke.

## Rennen

1. Erstellen Sie eine Klasse "Rennen".
2. Ein Rennen hat folgende Eigenschaften:
   * einen Namen,
   * die Anzahl der teilnehmenden Schnecken,
   * die teilnehmenden Schnecken selbst, als Array: einfachere Umsetzung mit konstanter Teilnehmerzahl,
   * die Länge der zu kriechenden Strecke.
3. Überlegen Sie sich, welche dieser Werte bereits im Konstruktor gesetzt werden sollten.
4. Legen Sie in der Klasse "Rennen" eine Methode "void addRennschnecke(Rennschnecke neueSchnecke)" an, welche dem Rennen eine Schnecke hinzufügt.
5. Legen Sie in der Klasse "Rennen" eine Methode "void removeRennschnecke(String name)" an, welche eine Schnecke aus dem Rennen entfernt.
6. Legen Sie in der Klasse "Rennen" eine Methode "public String toString()" an, welche die Daten des Rennens als String zurückgibt.
   * Tipp: Um die Daten der beteiligten Schnecken zurückzugeben, können Sie deren toString() Funktion benutzen.
7. Testen Sie Ihre Klasse vom Hauptprogramm aus!
8. Legen Sie in der Klasse "Rennen" eine Methode "Rennschnecke ermittleGewinner()" an, welche null zurückliefert, wenn noch keine der teilnehmenden Schnecken das Ziel erreicht hat und anderenfalls die Gewinnerschnecke zurückgibt.
9. Legen Sie in der Klasse "Rennen" eine Methode "void lasseSchneckenKriechen()" an, welche alle teilnehmenden Schnecken einmal kriechen lässt.
10. Legen Sie in der Klasse "Rennen" eine Methode "void durchfuehren()" an, welche so lange lasseSchneckenKriechen() aufruft, bis eine der Schnecken das Ziel erreicht hat.
    * Tipp: Ob eine Schnecke im Ziel angekommen ist, können Sie mit der Methode ermittleGewinner() herausfinden.

## Wettbüro

1. Erstellen Sie eine Klasse Wettbuero.
2. Ein Wettbuero hat die folgenden Eigenschaften:
   * Es weiß, für welches Rennen es seine Wetten entgegennimmt.
   * Es verfügt über eine Liste (z.B. ein Array) von angenommenen Wetten.
   * Es hat einen festen Faktor, mit welchem Wetteinsätze bei einem Gewinn multipliziert werden.
3. Legen Sie in der Klasse "Wettbuero" eine Methode "wetteAnnehmen(String schneckenName, int wettEinsatz, String spieler)" an, welche eine Wette entgegennimmt. Die Wette ist bezogen auf eine Schnecke für das Rennen, das von dem Büro bearbeitet wird.
   * Um die einzelnen Wetten speichern zu können, sollten ihre Daten in eigenen Objekten der Klasse "Wette" gespeichert werden. Erstellen Sie diese Klasse selbst.
   * Tipp: Denken Sie an toString().
4. Legen Sie in der Klasse "Wettbuero" eine Methode "rennenDurchfuehren()" an, welche das betreute Rennen durchführt.
5. Legen Sie in der Klasse "Wettbuero" eine Methode "toString()" an, welche die Daten des Wettbueros, die Daten des Rennens sowie sämtliche abgeschlossene Wetten als String zurückgibt.
6. Testen Sie Ihr Programm!
7. Tauschen Sie die Klasse Rennen mit einem Teilnehmer, der ebenfalls bereits fertig ist.
   * Sollten ihre Programme immer noch laufen?
   * Treten Fehler auf? Wenn ja: ändern Sie die Programme so ab, dass Sie problemlos Klassen austauschen können.

## Zusatz

Es gibt noch ein paar Dinge, die in unserem Wettbüro nicht so laufen wie sie sollten. Machen Sie sie besser.

* Dieselbe Schnecke kann in ein Rennen zweimal eingetragen werden.
* Es können negative Wetten abgeschlossen werden.
* Gehen zwei Schnecken gleichzeitig durchs Ziel, wird die Schnecke ausgegeben, auf die die Suche nach dem Gewinner zuerst stößt. Das ist ganz schön ungerecht.

# Amphibienfahrzeuge

In dieser Aufgabe soll ein Paket mit Fahrzeugen implementiert werden. Vorausgesetzt sei folgende Aufteilung eines Pakets fahrzeuge in Klassen: 

1. Erstellen Sie die Interfaces *Fahrzeug*, *Wasserfahrzeug*, *Landfahrzeug* und *Amphibienfahrzeug*.
2. Erstellen Sie die Klasse Auto. Implementieren Sie neben dem Konstruktor Auto(int anzahlRäder, double gewicht), der gewicht und anzahlRädersetzt, die Methoden double getGewicht(), die das Gewicht zurückliefert, und int getAnzahlRaeder(), die die Anzahl der Räder zurückliefert.
3. Erstellen Sie die Klasse Schiff. Implementieren Sie neben dem Konstruktor Schiff(double gewicht, double tiefgang), der gewicht und tiefgang setzt, zwei Methoden double getGewicht(), die das Gewicht zurückliefert, und double getTiefgang(), die den Tiefgang zurückliefert.
4. Erstellen Sie die Klasse Amphibie. Implementieren Sie neben dem Konstruktor Amphibie(double gewicht, double tiefgang, int anzahlRaeder) weitere evtl. notwendige Methoden. Die Methode double getTiefgang() gibt die Ausführung an die entsprechende Methode von Schiffweiter. Entscheiden Sie selbst wie Sie double getGewicht() implementieren.
5. Schreiben Sie in der main-Funktion Anweisungen, die ein Objekt der Klasse Amphibie mit Gewicht 3000.0, 6 Rädern und einem Tiefgang von 3.0 erzeugt, Lassen Sie sich anschließend das Gewicht, die Anzahl an Rädern und den Tiefgang ausgeben.

# Stack

Ein Kellerspeicher oder Stack ist ein "Gerät", das es erlaubt, Objekte der gleichen Art zu speichern. Die Objekte werden übereinander abgelegt. Sie werden von oben hineingelegt und ggf. auch wieder entnommen. Das Objekt, das zuerst gespeichert wurde, liegt also ganz unten, das zuletzt eingelegte ganz oben.

Beispiel: Sie sammeln alte Tageszeitungen in einem Pappkarton bis zum Abholtag. Falls der Karton in Breite und Länge der Zeitung entspricht, haben Sie einen klassischen Stack.

Etwas formaler betrachtet benötigt man für einen Stack folgende Operationen:

* push(obj) legt das Objekt *obj* im Stack ab,
* pop() entfernt das oberste Element und gibt seinen Wert zurück,
* top() rührt kein Objekt an, gibt aber den Wert des obersten Elementes zurück,
* isEmpty() rührt kein Objekt an und gibt als Antwort false, falls noch ein Objekt im   
   Stack ist, ansonsten true.

Beispiel: Mit push(sz070919) legen sie die Süddeutsche Zeitung vom 19.09.2007 in den Karton. Mit pop() holen Sie die zuletzt eingelegte Zeitung wieder heraus. Mit top() schauen Sie nach, welche Zeitung oben liegt. Mit isEmpty() können Sie prüfen, ob überhaupt noch eine Zeitung im Karton liegt.

Sei stackInt ein Kellerspeicher für Ganzzahlen. Folgende Operationen werden vorgenommen:

1. stackInt.push(4);
2. stackInt.push(9);
3. stackInt.push(3 \* stackInt.top() );
4. int x = 4 + stackInt.pop();
5. stackInt.push(x);

Geben Sie grafisch an, was nun in stackInt liegt:

# Geometrie Paket

Im Folgenden sehen Sie das Klassendiagramm eines Geometrie-Paketes. Wir wollen uns dieses nach und nach aufbauen.



Die Methoden sind durch ihre Namen weitaus selbst erklärend. Auf der nächsten Seite befinden sich Beschreibungen für ausgewählte Methoden, im Zweifelsfall fragen Sie ihren Hörsaalleiter.

Aufgaben:

1. Analysieren Sie das Klassendiagramm.
2. Implementieren Sie die Klassen in der Reihenfolge: 1. Punkt2D, 2. Figur, 3. Linie, 4. Gerade, 5. Strecke, 6. Kreis.

Versuchen Sie bei der Implementierung möglichst viel Code wiederzuverwenden. Falls Ihnen eine Methode auffällt, welche in einer anderen Klasse auch von Nutzen sein könnte, kann es sich lohnen die Implementierung an der anderen Stelle vorzunehmen und die Funktionalität an der ursprünglichen Stelle zu nutzen. (Bsp. Kreis::abstand(Punkt2D)).

|  |  |
| --- | --- |
| Klasse: | Punkt2D |
| Methoden: | Semantik: |
| betrag(): double | Liefert die Länge des Vektors (0, 0) 🡪 (X, Y). |
| plus(p: Punkt2D): Punkt2D | Addiert p zur aktuellen Koordinate. |
| mal(faktor: double): Punkt2D | Multipliziert X und Y mit faktor. |
| skalarProdukt(p: Punkt2D): double | Liefert Skalarprodukt mit p. |
| orthogonal(): Punkt2D | Liefert eine Koordinate (X‘, Y‘), sodass der Vektor  (0, 0) 🡪 (X‘, Y‘) senkrecht auf dem aktuellen Vektor (0, 0) 🡪 (X, Y) steht. |

|  |  |
| --- | --- |
| Klasse: | Figur |
| Methoden: | Semantik: |
| verschiebeNach(p: Punkt2D): void | Verschiebt die Figur auf den neuen Punkt p. |
| verschiebeUm(p: Punkt2D): void | Verschiebt die Figur um den in p angegebenen X- und Y-Wert. |