

Aufgabenblatt 1

Java: Einführung, Objektorientierte Programmierung

Wichtige Ankündigungen

- Das Vorlesungsmaterial, die Übungsblätter und die Vorlagen für die Hausaufgaben finden Sie unter <https://git.tu-berlin.de/algodat-sose25/material.git>.
- Alle Übungen sind in Einzelarbeit zu erledigen. Kopieren Sie niemals Code und geben Sie Code in keiner Form weiter.
- Es gibt drei Aufgabentypen:
 - **Tutorium** → Besprechung im Tutorium
 - **Hausaufgabe** → Eigenarbeit mit Hilfe in Rechnerübungen
 - **Klausurvorbereitung** → Optionale Klausurübung

Abgabe (bis 19.05.2025 23:59 Uhr)

Die folgenden Dateien müssen für eine erfolgreiche Abgabe im git Ordner eingecheckt sein:

Geforderte Dateien:

Blatt01/src/Pair.java	Aufgabe 2
Blatt01/src/Polygon.java	Aufgabe 3
Blatt01/src/ConvexPolygon.java	Aufgabe 3
Blatt01/src/Tetragon.java	Aufgabe 3
Blatt01/src/Triangle.java	Aufgabe 3
Blatt01/src/RegularPolygon.java	Aufgabe 3

Als Abgabe wird jeweils nur die letzte Version im main branch in git gewertet.

Aufgabe 1: Objektorientierte Programmierung (Tutorium)

Diese Aufgabe soll Ihnen die Grundprinzipien der Objektorientierten Programmierung näher bringen, welche Sie auch in der Hausaufgabe brauchen. Die Aufgabe wird im Tutorium gemeinsam erarbeitet. Alternativ (oder zusätzlich) können Sie sich das Videotutorium anschauen und dabei die Aufgabenteile selbstständig lösen, indem Sie die Videos pausieren. Sollte Ihnen das noch schwer fallen, schauen Sie sich die Videos komplett an und bearbeiten Sie die Aufgabenteile danach alleine, indem Sie umsetzen, was Sie gerade gelernt haben.

- 1.1 Definieren Sie Klassen und Objekte.
- 1.2 Implementieren Sie eine Klasse *KegeLrobbe* und testen Sie Ihre Implementation in einer *main*-Methode.
- 1.3 Erstellen Sie eine Hierarchie von *Wirbeltieren* am Beispiel von zwei Klassen von Wirbeltieren mit jeweils zwei Vertretern dieser Klassen.
- 1.4 Wenn Sie nun diese Hierarchie implementieren mit *Wirbeltiere* als abstrakte Klasse. Überlegen Sie sich Attribute, die Sie den Klassen geben könnten. Wie sähe der Code dafür aus, wenn Sie in jeder Klasse nur den Konstruktor implementieren? Welche dieser Attribute werden unter welchen Bedingungen vererbt?
- 1.5 Wie testen Sie die Funktionalität Ihrer Klassen? Wie erstellen Sie einen Array von *Wirbeltiere*? Können Sie unterschiedliche Tiere in diesem Array speichern? Wenn ja, wie? Wie funktioniert Casting an diesem Beispiel?
- 1.6 Was ist Polymorphie? Implementieren Sie eine abstrakte Methode *essen()*.

Bemerkung zu Aufgaben 2 und 3

Die Hausaufgaben dieses Blattes sind als Annäherung an die Programmierung in Java gedacht. Die Beschreibung der Aufgaben ist lang im Vergleich zu dem Code, der geschrieben werden soll, um Ihnen einen guten Einstieg zu ermöglichen, auch wenn Sie noch keine oder wenig Erfahrung mit Java haben. In den folgenden Aufgabenblättern werden die Anforderungen deutlich steigen. Nutzen Sie dieses Einstiegsblatt dazu, möglichst viel Routine in den Grundfertigkeiten der Programmierung in Java und in dem Umgang mit der IDE IntelliJ IDEA zu entwickeln.

Aufgabe 2: Generics (Hausaufgabe) (30 Punkte)

Implementieren Sie eine Klasse *Pair*, die zwei Elemente eines beliebigen (*generischen*) Typs speichern kann. Die beiden Elemente müssen denselben Typ besitzen, also Elemente derselben Klasse sein. Sie sollen als private Variablen in der Klasse *Pair* gespeichert werden. Der Zugriff geschieht über sogenannte *getter* und *setter* Methoden, siehe API.

API eines Paares

`public class Pair<E>`

<code>Pair(E first, E second)</code>	erzeugt ein Paar mit den beiden Elementen
<code>Pair(Pair<E> other)</code>	Copy Konstruktor: erzeugt eine Kopie des übergebenen Paares
<code>void swap()</code>	vertauscht die beiden Elemente
<code>E getFirst()</code>	gibt das erste Element zurück
<code>void setFirst(E first)</code>	speichert das übergebene Argument als erstes Element des Paares
<code>E getSecond()</code>	gibt das zweite Element zurück
<code>void setSecond(E second)</code>	speichert das übergebene Argument als zweites Element des Paares

Außerdem sollen die von der `Object`-Klasse geerbten Methoden `equals()` und `toString()` mit spezifischen Methoden überschrieben werden. Dabei soll `equals` die *semantische* Gleichheit eines `Pair`-Objektes mit einem anderen Objekt überprüfen (siehe Abschnitt „Syntaktische und Semantische Gleichheit von Objekten“ in dem Skript *Kleine Einführung in Java*).

Die Methode `toString()` soll so implementiert werden, dass der Befehl

```
System.out.println(new Pair<>(21, 84));
```

zu der Ausgabe

```
Pair<21, 84>
```

führt. **Hinweise:**

Sie können und sollen bei dieser Aufgabe ausgiebigen Gebrauch der automatischen Generierung von Methoden von IntelliJ IDEA machen. Dazu muss der Cursor innerhalb der Klasse `Pair` positioniert sein. Hier können Sie das **Generate**-Menü anzeigen lassen, indem Sie einen Rechtsklick auf eine freie Stelle im Codeeditor ausführen und im Kontextmenü **Generate** auswählen. Noch schneller kommen Sie mit der Tastenkombination `Alt+Inser` zu diesem Menü. Auf diese Weise können Sie den ersten Konstruktor, die getter- und setter-Methoden, `toString()` sowie `equals()` erzeugen. Vor der Generierung des Konstruktors sollten Sie die beiden privaten Objektvariablen definiert haben und zur Einbindung in den Konstruktor auswählen. Die automatisch erzeugte `toString()` Methode müssen Sie ein wenig anpassen, damit sie das vorgegebene Format der Ausgabe erfüllt. Beim Erzeugen der `equals()` Methode wird immer auch die `hashCode()` Methode erzeugt. Diese können Sie einfach ignorieren. Die Bedeutung der Methode und das wichtige Zusammenspiel mit `equals()` wird in der Vorlesung und Übung zu Hashtabellen besprochen.

Auch wenn Sie denken, dass es zu Übungszwecken sinnvoller sein könnte, die Methoden selbst zu schreiben, empfehlen wir nachdrücklich in dieser Aufgabe, die automatische Generierung zu verwenden. Diesen Automatismus in den Arbeitsablauf zu integrieren, wird bei späteren Programmieraufgaben helfen, Zeit zu sparen und vielleicht auch Fehler zu vermeiden. Sie können hier natürlich auch zunächst eigene Methoden programmieren, auskommentieren, und dann dieselben Methoden generieren lassen und mit den eigenen vergleichen.

Das vorgegebene Gerüst `Pair.java` enthält eine `main()` Methode, die Sie zum Testen verwenden können. Die erwartete Ausgabe steht im Kommentar der `main()` Methode. Damit die *semantische Gleichheit* den korrekten Wert `true` ergibt, muss die `equals()` Methode korrekt implementiert sein, und damit die Variable `pair2b` am Ende den richtigen Wert hat, muss der Copy Konstruktor korrekt

implementiert sein. Die `main()` Methode stellt keinen vollständigen Test aller Funktionalitäten der Klasse `Pair` dar.

Hinweis:

- Es wird Ihnen auffallen, dass das Programm, trotz Implementieren der Funktionen, Fehler beim Kompilieren erzeugt. Dies hängt damit zusammen, dass der komplette `src` Ordner kompiliert wird und nicht nur eine Datei. Sie haben mehrere Wege, um dies zu umgehen. Sie können z.B. in die Einstellungen gehen, dann zu **Compiler | Excludes** und dort die jeweiligen unvollständigen java Dateien entfernen. Bitte beachten Sie, dass Sie diese Sonderregelungen dann später beim Bearbeiten der nächsten Aufgaben auch wieder entfernen müssen.

Aufgabe 3: Vererbung (Hausaufgabe)

In dieser Aufgabe soll eine Hierarchie geometrischer Formen implementiert werden. Dabei sind die geometrischen Objekte in ihrer Größe und ihrer Lage im zwei-dimensionalen Raum definiert. Die Hierarchie ist in Abbildung 1 dargestellt. Auf der obersten Abstraktionsstufe steht die Schnittstelle (*interface*) `Shape`. Sie definiert, welche Funktionalitäten alle geometrischen Formen der Hierarchie bereitstellen müssen. Hier sind das nur die beiden Methoden `perimeter()` und `area()`, die den Umfang und den Flächeninhalt der jeweiligen Form zurückgeben. Auf der nächsten Stufe wird die abstrakte Klasse `Polygon` definiert. Alle von ihr abgeleiteten Klassen stellen Formen dar, die durch ihre Eckpunkte (*vertices*) definiert sind. Diese beiden Klassen sind als Dateien `Shape.java` und `Polygon.java` vorgegeben, die Sie für diese Aufgabe nicht verändern dürfen. Desweiteren ist die Klasse `Vector2D` vorgegeben, die Ortsvektoren bzw. Punkte im zweidimensionalen Raum definiert.

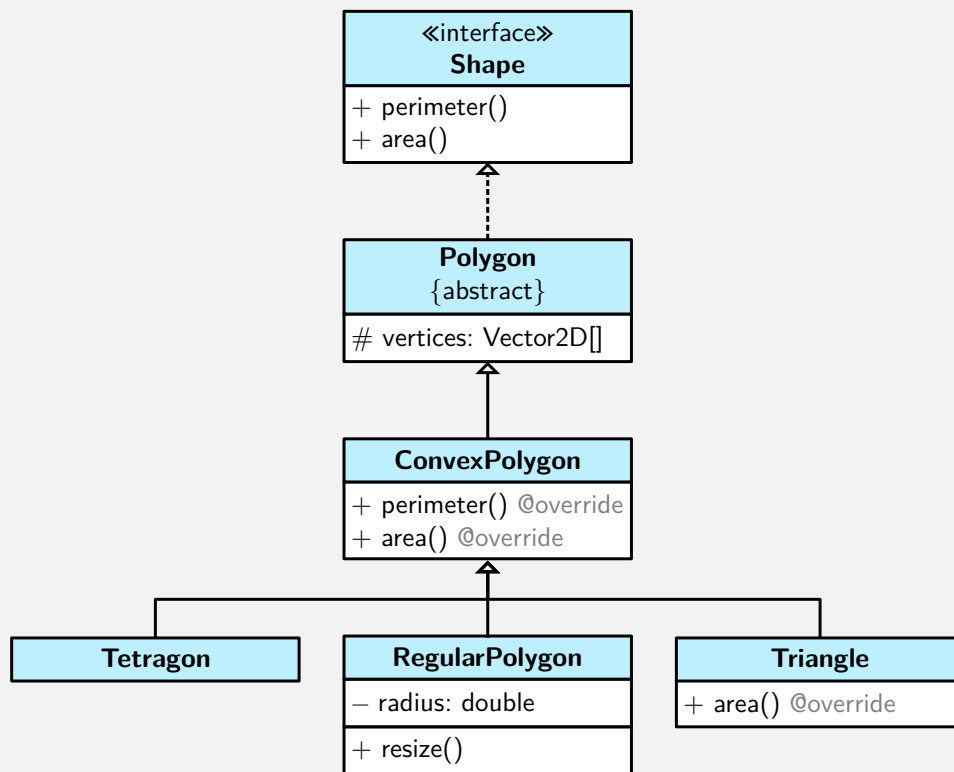
Ihre Aufgabe besteht darin, gemäß der abgebildeten Hierarchie, die vier konkreten geometrischen Formen *konvexe Polygone*, *regelmäßiges Polygon*, *Viereck* und *Dreieck* als Klassen zu implementieren. Die Anforderungen sind in den folgenden Teilaufgaben genauer spezifiziert.

3.1 Konvexe Polygone (Klasse `ConvexPolygon`) (25 Punkte)

Das geometrische Objekt wird gemäß der Vorgabe aus der abstrakten Klasse `Polygon` durch die Eckpunkte in einem `Vector2D`-Array dargestellt, das in der entsprechenden Objektvariable `vertices` gespeichert wird. Die Funktionen müssen nur unter der Annahme funktionieren, dass die in `vertices[]` gegebenen Punkte umlaufend sind, also im oder gegen den Uhrzeigersinn geordnet. Die Klasse `ConvexPolygon` soll folgende Methoden implementieren. Überlegen Sie anhand der Beschreibung der Methoden gegebenenfalls, welche Parameter jeweils übergeben werden müssen und welche Rückgabedatentypen gewählt werden müssen.

- ein Konstruktor, dem ein Array aus Eckpunkten übergeben wird. Es braucht *nicht* überprüft zu werden, ob die übergebenen Punkte tatsächlich ein *konvexes* Polygon darstellen.
- die von der Schnittstelle `Shape` geforderten Methoden `area()` und `perimeter()`, die den Flächeninhalt beziehungsweise den Umfang des konvexen Polygons berechnet. Zur Berechnung des Flächeninhalts empfiehlt sich eine Triangulierung, so dass die Flächeninhalte der Dreiecke per `Triangle.area()` berechnet und addiert werden können. Eine Art der Unterteilung in Dreiecke wird in Abbildung 2 gezeigt. Für Dreiecke wird `area()` in der nächsten Teilaufgabe implementiert. Diese einfache Art der Triangulierung funktioniert nur für *konvexe* Polygone. Wie oben beschrieben, brauchen Sie nicht zu überprüfen, ob das Polygon wirklich konvex ist.
- eine Methode mit der Signatur

Abbildung 1: UML die Hierarchie geometrischer Formen



```
public static double totalArea(Polygon[] polygons)
```

Die Methode soll den gesamten Flächeninhalt aller übergebenen Polygone zurückgeben.

- eine Methode mit der Signatur

```
public static Polygon[] somePolygons()
```

Die Rückgabe dieser Methode soll ein Array bestehend aus den folgenden vier Objekten (in der gelisteten Reihenfolge) sein:

- Dreieck mit Eckpunkten (0, 0), (10, 0) und (5, 5)
 - Viereck mit Eckpunkten (0, 0), (10, -5), (12, 2) und (3, 17)
 - Regelmäßiges Fünfeck mit Radius 1
 - Regelmäßiges Sechseck mit Radius 1
- Überschreiben Sie die Methode toString() so, dass der Befehl

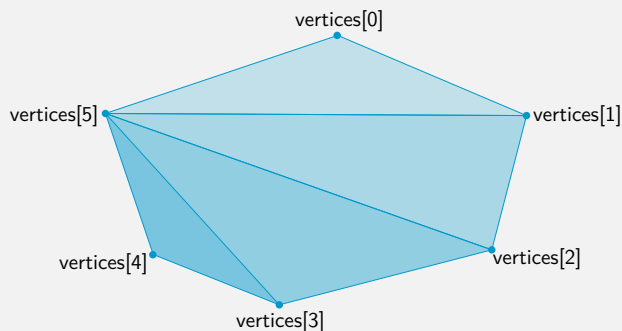
```
Vector2D a = new Vector2D(0, 0);
Vector2D b = new Vector2D(10, 0);
Vector2D c = new Vector2D(5, 5);
Polygon polygon = new ConvexPolygon(new Vector2D[] {a, b, c});
System.out.println(polygon);
```

zu der Ausgabe

```
ConvexPolygon([(0.0, 0.0), (10.0, 0.0), (5.0, 5.0)])
```

führt. Tipp: Nutzen Sie *Generate > toString()* und wählen Sie die Variable `vertices` aus. Dadurch wird eine Implementation erzeugt, die Sie nur noch ein bisschen anpassen müssen.

Abbildung 2: Triangulierung eines konvexen Polygons



Hier am Beispiel eines Sechsecks:
Eine Ecke des Polygons wird festgehalten, hier `vertices[5]`, und ist eine Ecke von allen Dreiecken. Die anderen beiden Ecken sind zwei weitere benachbarte Ecken des Polygons (`vertices[n]` und `vertices[n+1]` für `n` von 0 bis 3).

3.2 Dreiecke (Klasse **Triangle**) (20 Punkte)

Die Klasse **Triangle** soll folgende Methoden implementieren

- Mindestens die beiden folgenden Konstruktoren:

```
public Triangle(Vector2D a, Vector2D b, Vector2D c) {  
    public Triangle(Triangle triangle)
```

Der erste erstellt ein **Triangle** Objekt mit den drei gegebenen Eckpunkten, und der zweite ist ein Copy-Konstruktor. Er erstellt ein **Triangle** Objekt, das die (syntaktisch) gleichen Eckpunkte besitzt, wie das übergebene Dreieck.

- die `area()`-Methode soll überschrieben werden. Sie soll den Flächeninhalt des Dreiecks zurückgeben. Der Flächeninhalt eines Dreiecks bei gegebenen Eckpunkten kann zum Beispiel mit der *Heron Formel* bestimmt werden. Tipps: `Vector2D` implementiert die Methode `length()` und enthält eine Implementation der Heron Formel in der `main()` Methode, die Sie verwenden dürfen.
- Überschreiben Sie die `toString()`-Methode so, dass ein Aufruf von

```
// sei t ein Variable vom Typ Triangle  
// mit den Punkten (0.0,0.0), (10.0,0.0), (5.0,5.0)  
system.out.println(t);
```

zur Ausgabe

```
Triangle([(0.0, 0.0), (10.0, 0.0), (5.0, 5.0)])
```

führt. Siehe dazu auch das Beispiel in der main-Methode in `Triangle.java`

3.3 Vierecke (Klasse `Tetragon`) (5 Punkte)

Die Klasse `Tetragon` braucht nur einen Konstruktor mit der Signatur

```
public Tetragon(Vector2D a, Vector2D b, Vector2D c, Vector2D d) {
```

zu besitzen. Ansonsten genügen die geerbten Methoden.

3.4 Regelmäßige Polygone (Klasse `RegularPolygon`) (20 Punkte)

Die Klasse `RegularPolygon` stellt regelmäßige Polygone dar, deren Mittelpunkt im Ursprung $(0, 0)$ liegt und deren Eckpunkte auf dem Kreis mit dem Umkreisradius `radius` (Abstand vom Mittelpunkt zu jedem Eckpunkt) liegen. Sie sind durch die Anzahl der Ecken `N` und den Umkreisradius `radius` eindeutig definiert. Implementieren Sie einen entsprechenden Konstruktor sowie einen Copy-Konstruktor

```
public RegularPolygon(int N, double radius)
public RegularPolygon(RegularPolygon polygon)
```

sowie die Methode

```
public void resize(double newradius)
```

die es erlaubt den Radius eines Objektes zu ändern. Denken Sie daran, dass dabei auch die Eckpunkte in der geerbten Variable `vertices` entsprechend verändert werden müssen.

Zur Berechnung der Koordinaten der Eckpunkte eignet sich die *Berechnung am Einheitskreis*. Zunächst muss der Winkel zwischen den Punkten vom Ursprung aus berechnet werden, was $360^\circ/\text{\#Ecken}$ entspricht. Setzt man diesen Winkel für jeden Eckpunkt in Sinus und Cosinus ein, bekommt man die Koordinaten für `Radius=1`. Das funktioniert, weil die Strecke vom Ursprung zu einem Eckpunkt die Hypotenuse eines rechtwinkligen Dreiecks bildet. Die Koordinaten müssen anschließend noch mit dem gegebenen Radius skaliert werden.

Fragen zur Diskussion im Tutorium:

- Könnte `Shape` auch als Klasse oder abstrakte Klasse definiert werden?
- Könnte `Polygon` auch als Interface definiert werden?
- In welchen Fällen sind Copy-Konstruktoren wichtig?

Aufgabe 4: Laufzeitorientierung (Klausurvorbereitung)

Das Two-Sum-Problem ist ein typisches Problem, das Ihnen z.B. in einem Vorstellungsgespräch gestellt werden könnte:

Es sind ein Array und eine Zahl `k` gegeben, beide vom Typ `int`. Schreiben Sie eine Methode, welche prüft, ob es in dem Array ein Paar gibt, welches in der Summe `k` ergibt.

4.1 Geben Sie ein Beispiellarray an, für den diese Methode für `k = 3` `true` oder `false` ausgibt.

4.2 Geben Sie die Brute-force Lösung des Problems und deren Laufzeit an.

4.3 Überlegen Sie sich eine schnellere Lösung, indem Sie den Array sortieren und dann von vorne und hinten gleichzeitig anfangen.

Die Lösung wird in dem Videotutorium 1 als Video bereitgestellt. Versuchen Sie erst selbst auf die Lösung zu kommen, aber mindestens sie danach selbst zu rekonstruieren.

Was Sie nach diesem Blatt wissen sollten:

- was Objekte und Klassen sind.
- wie man Konstruktoren schreibt, Objekte erzeugt und Methoden auf diesen Objekten aufruft.
- wie Sie for-Schleifen und if-Bedingungen verwenden.
- was getter- und setter-Methoden sind und wie man sie schreibt.
- wie Sie Arrays von Objekten erstellen und über diese iterieren.
- was eine abstrakte Klasse ist und wie Klassen erben.
- wie Sie eine `main()`-Methode schreiben und ausführen.
- wie Sie Ihre geschriebenen Methoden selbst ausprobieren und Kontrollvariablen ausgeben.