Aufgabenblatt 1

Java Einführung I - Vererbung

Wichtige Ankündigungen

- **Erinnerung Prüfungsanmeldung**: (für die meisten: in Moses) Deadline ist am **26.05.2023**. Ohne Prüfungsanmeldung können Sie nicht an der Klausur teilnehmen und bekommen keine Prüfungsleistungen angerechnet.
- Registrieren Sie sich unter `http://testing.neuro.tu-berlin.de/index.php`, wenn Sie das noch nicht getan haben.
- Das Vorlesungsmaterial, die Übungsblätter und die Vorlagen für die Hausaufgaben finden Sie unter `https://git.tu-berlin.de/algodat-sose23/Material.git`.
- Alle Übungen sind in Einzelarbeit zu erledigen. Kopieren Sie niemals Code und geben Sie Code in keiner Form weiter. Die Hausaufgaben sind Teil Ihrer Prüfungsleistung. Finden wir ein Plagiat (wir verwenden Plagiatserkennungssoftware), führt das zum Nichtbestehen des Kurses.

Abgabe (bis 08.05.2023 23:59 Uhr)

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

Geforderte Dateien:	
`Blatt01/src/Pair.java`	Aufgabe 3
`Blatt01/src/Polygon.java`	Aufgabe 4
`Blatt01/src/ConvexPolygon.java`	Aufgabe 4
`Blatt01/src/Tetragon.java`	Aufgabe 4
`Blatt01/src/Triangle.java`	Aufgabe 4
`Blatt01/src/RegularPolygon.java`	Aufgabe 4

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

Aufgabe 1: Laufzeitoptimierung (Klausurvorbereitung)

Das Two-Sum-Problem ist ein typisches Problem, was 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.

1.1 Geben Sie je ein Beispielarray an, für den diese Methode für $k = 3$ `true` oder `false` ausgibt.

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

1.3 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.

Aufgabe 2: OOP (Klausurvorbereitung)

Diese Aufgabe soll Ihnen die Grundprinzipien der Objektorientierten Programmierung näher bringen, welche Sie auch in der Hausaufgabe brauchen. Schauen Sie sich das Videotutorium an und lösen Sie dabei die Aufgabenteile selbstständig, 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. Die Aufgabe wird auch im Online-Tutorium behandelt.

2.1 Definieren Sie Klassen und Objekte.

2.2 Implementieren Sie eine Klasse `Kegelrobbe` und testen Sie Ihre Implementation in einer `main`-Methode.

2.3 Erstellen Sie eine Hierachie von *Wirbeltieren* am Beispiel von zwei Klassen von Wirbeltieren mit jeweils zwei Vertretern dieser Klassen.

2.4 Wenn Sie nun diese Hierarchie implementieren mit `Wirbeltiere` als abstrakte Klasse, wie sähe der Code dafür aus, wenn Sie in jeder Klasse nur den Konstruktor implementieren? Überlegen Sie sich Attribute, die Sie den Klassen geben könnten. Welche dieser Attribute werden unter welchen Bedingungen vererbt?

2.5 Wie testen Sie die Funktionalität Ihrer Klassen? Wie erstellen Sie einen Array von `Vertebrata`? Können Sie unterschiedliche Tiere in diesen Array schreiben? Wenn ja, wie? Wie funktioniert Casting an diesem Beispiel?

2.6 Was ist Polymorphie? Implementieren Sie eine abstrakte Methode `essen()`.

Bemerkung zu Aufgaben 3 und 4

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 erstellt 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 3: 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 der Klasse `Pair` gespeichert werden. Der Zugriff geschieht über sogenannte *getter* und *setter* Methoden, siehe API.

API eines Paares

public class Pair<E>

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

Die von der `Object`-Klasse geerbten Methoden `equals()` und `toString()` sollen 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 zu machen. Dazu muss der Cursor innerhalb der Klasse `Pair` positioniert sein und Sie drücken CTRL+N, bzw. CMD+N, oder klicken die rechte Maustasten und wählen *Generate* im Kontext-Menu aus. 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()` Methoden 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 Method 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 erwartet 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 richtig Wert hat, muss der Copy Konstruktor korrekt implementiert sein. Die `main()` Methode stellt keinen vollständigen Test aller Funktionalitäten der Klasse `Pair` dar.

Aufgabe 4: 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 geomtrischen 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. Weiterhin wäre zum Beispiel eine Methode zur Visualisierung denkbar. 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 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.

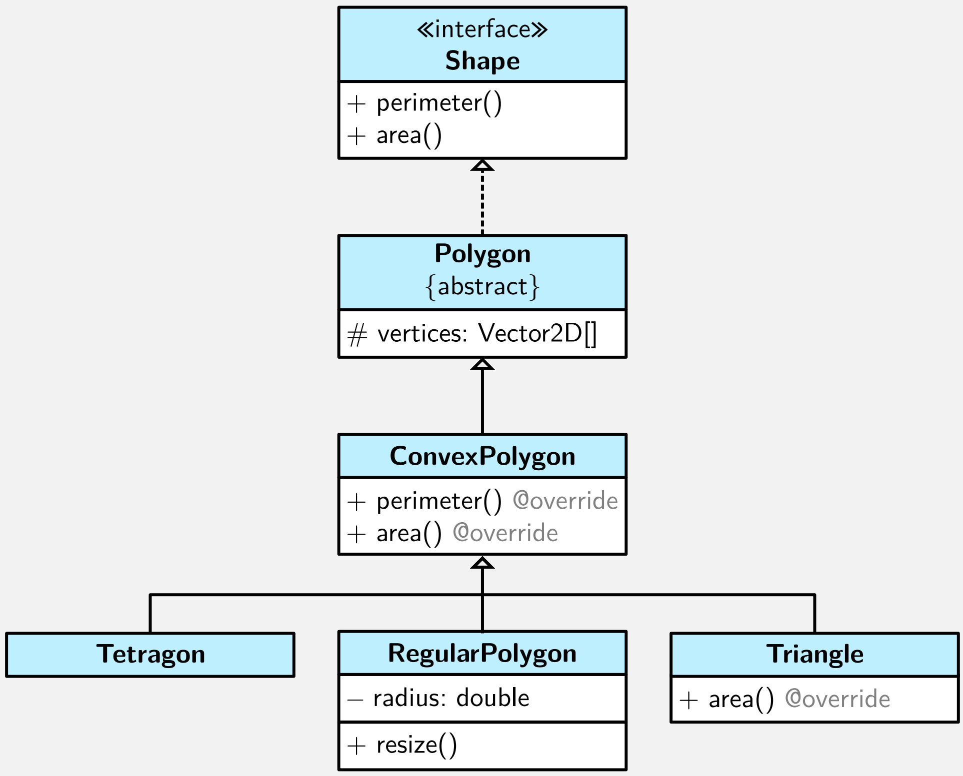
4.1 Konvexe Polygone (Klasse `ConvexPolygon`) (25 Punkte)

Die Klasse `ConvexPolygon` implementiert die von der Schnittstelle `Shape` geforderten Methoden. Das geometrische Objekt wird gemäß der Vorgabe aus der abstrakten Klasse `Polygon` durch die Eckpunkte in einem `Vector2D`-Array dargestellt und in der entsprechenden Objektvariable `vertices` gespeichert. Es braucht *nicht* überprüft zu werden, ob die übergebenen Punkte tatsächlich ein *konvexes* Polygon darstellen. Die Funktionen müssen außerdem nur unter der Annahme funktionieren, dass die in `vertices[]` gegebenen Punkte umlaufend sind, also im oder gegen den Uhrzeigersinn geordnet.

Der Flächeninhalt kann dadurch bestimmt werden, dass das Polygon in (disjunkte) Dreiecke zerlegt wird und deren Flächeninhalte aufsummiert werden, siehe Anhang. (Hier kann vorausgesetzt werden, dass das Objekt in der Tat ein *konvexes* Polygone darstellt.) Für Dreiecke wird `area()` in der nächsten Teilaufgabe implementiert.

Überschreiben Sie die Methode `toString()` so, dass der Befehl

Abbildung 1: UML die Hierarchie geometrischer Formen




```java
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.

Implementieren Sie die folgenden beiden Methoden:

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

Die Rückgabe von `somePolygons()` 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

Die Methode `totalArea()` soll den aufsummierten Flächeninhalt aller übergebenen Polygone zurückgeben.

4.2 Dreiecke (Klasse `Triangle`) (20 Punkte)

Die Klasse `Triangle` muss (mindestens) die folgenden beiden Konstruktoren besitzen:

```java
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.

Des Weiteren wird die `area()` Methode überschrieben und gibt den Flächeninhalt des Dreiecks zurück. Zur Berechnung kann z. B. die Formel von Heron verwendet werden. (Tipp: `Vector2D` implementiert die Methode `length()`.)

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

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

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

zu besitzen. Ansonsten genügen die geerbten Methoden.

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

Die Klasse `RegularPolygon` stellt regelmäßige Polygon dar, deren Mittelpunkt im Ursprung $(0, 0)$ liegt. Sie sind durch die Anzahl der Ecken und Ihren Radius eindeutig definiert. Implementieren Sie einen entsprechenden Konstruktor, sowie einen Copy-Konstruktor

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

sowie die Methode

```java
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.

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?

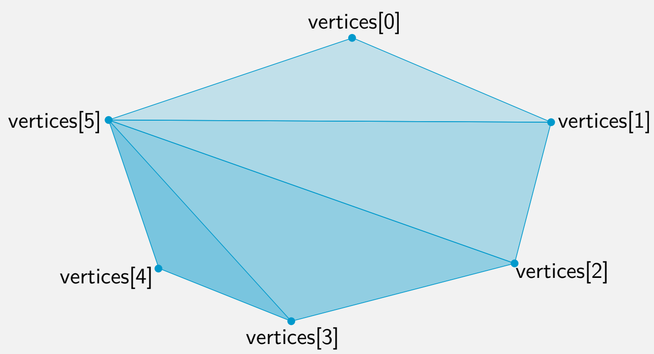
Was Sie nach diesem Blatt wissen sollten:

- 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.

Anhang zu Aufgabe 4

Zur Berechnung des Flächeninhalts eines konvexen Polygons in Aufgabe 4.1 empfiehlt sich eine Triangulierung, so dass die Flächeninhalte der Dreiecke per `Triangle.surface()` berechnet und addiert werden können. Eine Art der Unterteilung in Dreiecke wird in Abbildung 2 gezeigt.

Abbildung 2: Triangulierung eines konvexen Polynoms



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).

Diese einfache Art der Triangulierung funktioniert nur für *konvexe* Polygone. Wie in Aufgabe 4 beschrieben, brauchen Sie nicht zu überprüfen, ob das Polygon wirklich konvex ist.

Der Flächeninhalt eines Dreiecks bei gegebenen Eckpunkten kann zum Beispiel mit der *Heron Formel* bestimmt werden. Eine Implementation ist übrigens in der `main()` Methode von `Vector2D` versteckt, die Sie benutzen dürfen.

Zur Berechnung der Koordinaten der regulären Polygone in Aufgabe 4.4 eignet sich die 'Berechnung am Einheitskreis'. Zunächst muss der Winkel zwischen den Punkten vom Ursprung aus berechnet werden, was 360°/#Ecken entspricht. Setzt man diesen Winkel in Sinus und Cosinus ein, bekommt man die Koordinaten für Radius=1. Das funktioniert, weil die Strecke vom Ursprung zu unserem Punkt eine Hypothenuse eines rechtwinkligen Dreiecks an der X-Achse darstellt. Die Koordinaten müssen aber noch zum gegebenen Radius skaliert werden.