**Objektorientierte Programmierung I**
**freiwillige Übungsaufgaben**
**14.06.2022**

---

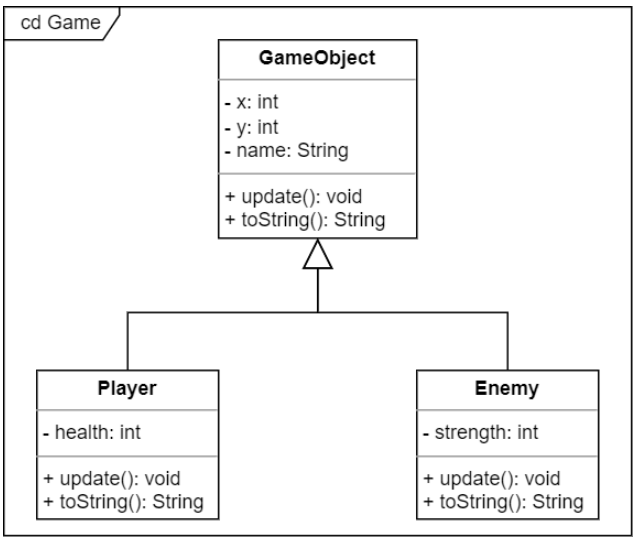
**Legende:**

🌶 = einfach
🌶🌶 = mittel
🌶🌶🌶 = schwer

---

## 🌶🌶 Aufgabe 1

Setze das folgende UML-Klassendiagramm als Java-Quellcode um:




Im Diagramm sind keine Konstrukturen, keine Getter und keine Setter dargestellt. Überlege also selbst, welche Konstruktoren sinnvoll wären und welche Getter und Setter du benötigst. Annotiere jede Methode, die eine andere Methode überschreibt, mit `@Override`.

Die `update()`-Methode soll…

- …in der Klasse `GameObject` gar nichts tun (leerer Methodenkörper).
- …in der Klasse `Player` dafür sorgen, dass die x-Koordinate um den Wert 1 erhöht wird.
- …in der Klasse `Enemy` dafür sorgen, dass sowohl die x- als auch die y-Koordinate jeweils um 1 erhöht werden.

Die `toString()`-Methode soll…

- …in der Klasse `GameObject` den Namen sowie die Koordinaten als String zurückgeben, z. B. in der folgenden Form:
  `"Name" @ (4,4)`
- …in der Klasse `Player` die Attribute in der folgenden Form als String zurückgeben:
  `Player "Super Mario" @ (4,4) (health = 100)`
- …in der Klasse `Enemy` die Attribute in der folgenden Form als String zurückgeben:
  `Enemy "Bowser" @ (8,9) (strength = 120)`

Überlege, wie du bei der Implementierung Code-Doppelungen vermeiden kannst (*Hinweis: Aufruf der Elternmethode*).

---

Teste deine Implementierung mit dem folgenden Hauptprogramm:

```
class Main {

    public static void main(String[] args) {
        GameObject[] gameObjects = {
            new Player(3, 4, "Super Mario", 100),
            new Enemy(7, 8, "Bowser", 120),
            new Enemy(1, 2, "Monty Mole", 30),
            new Enemy(-3, 4, "Thwimp", 45)
        };

        for (int i = 1; i <= 3; ++i) {
            System.out.println("Spielzug " + i + ":");
            for (GameObject gameObject : gameObjects) {
                gameObject.update();
                System.out.println("  " + gameObject);
            }
        }
    }

}
```

Wenn deine Implementierung korrekt ist, sollte die folgende Ausgabe entstehen:

```
Spielzug 1:
  Player "Super Mario" @ (4,4) (health = 100)
  Enemy "Bowser" @ (8,9) (strength = 120)
  Enemy "Monty Mole" @ (2,3) (strength = 30)
  Enemy "Thwimp" @ (-2,5) (strength = 45)
Spielzug 2:
  Player "Super Mario" @ (5,4) (health = 100)
  Enemy "Bowser" @ (9,10) (strength = 120)
  Enemy "Monty Mole" @ (3,4) (strength = 30)
  Enemy "Thwimp" @ (-1,6) (strength = 45)
Spielzug 3:
  Player "Super Mario" @ (6,4) (health = 100)
  Enemy "Bowser" @ (10,11) (strength = 120)
  Enemy "Monty Mole" @ (4,5) (strength = 30)
  Enemy "Thwimp" @ (0,7) (strength = 45)
```

**Objektorientierte Programmierung I**
**freiwillige Übungsaufgaben**
**14.06.2022**

---

### 🌶 Aufgabe 2

Schreibe eine Klasse `Point` mit den öffentlichen Attributen `x` und `y` (jeweils vom Typ `int`). Implementiere einen oder mehrere passende Konstruktoren. Überlade sodann die `equals()`-Methode, sodass es damit möglich wird, zwei Instanzen dieser Klasse auf Gleichheit zu überprüfen.

Die `equals()`-Methode erwartet als Parameter ein Objekt vom Typ `Object`. Um es für einen Vergleich nutzen zu können, musst du es zuerst in den Typ `Point` umwandeln. Die Methode muss also die folgende Form haben:

```
@Override
public boolean equals(Object obj) {
    Point otherPoint = (Point)obj;
    return /* hier Überprüfung einfügen */;
}
```

Du kannst deine Implementierung mit dem folgenden Hauptprogramm testen:

```
class Main {

    public static void main(String[] args) {
        Point a = new Point(1, 2);
        Point b = new Point(1, 2);
        System.out.println("Sind a und b gleich? " + a.equals(b));
        b = new Point(3, 4);
        System.out.println("Sind a und b gleich? " + a.equals(b));
    }

}
```

Was passiert, wenn du ein `Point`-Objekt mit einem Objekt eines anderen Typs vergleichst (z. B. `String`)? Kannst du das Verhalten erklären?

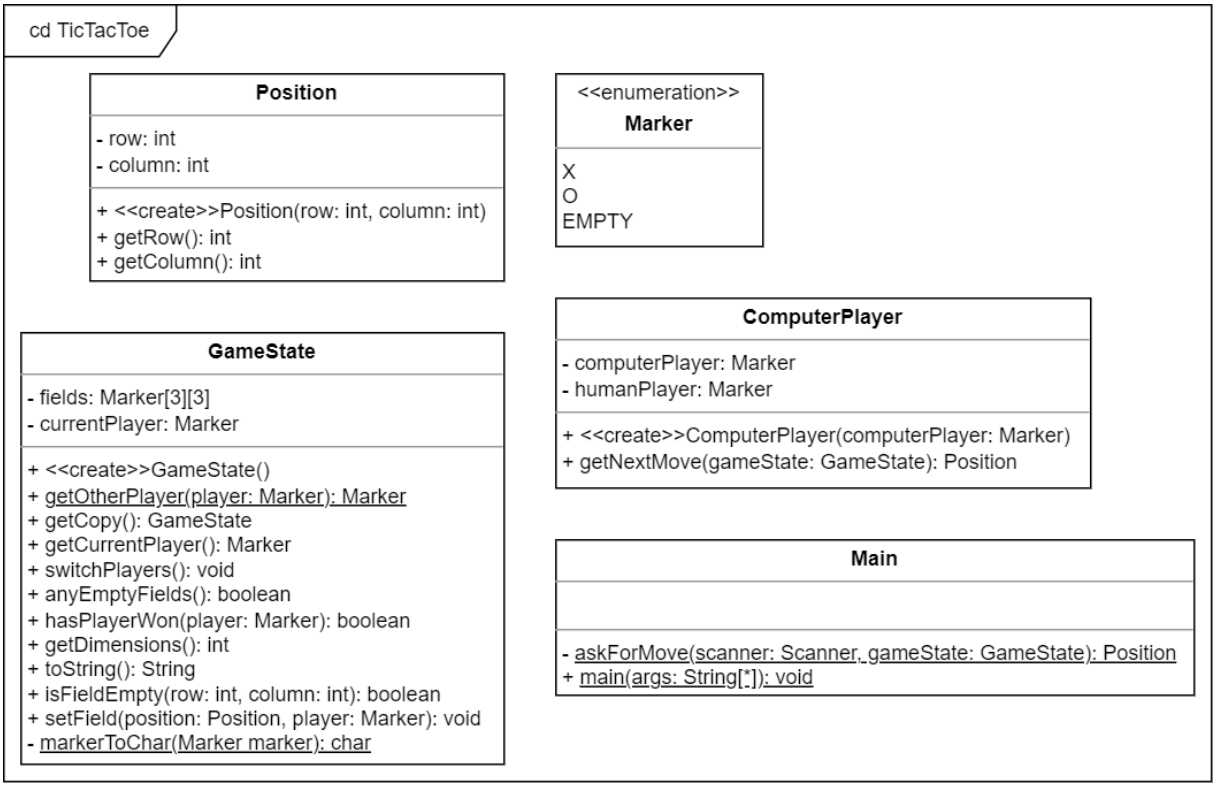
Dokumentation der `equals()`-Methode: https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals(java.lang.Object)

**Objektorientierte Programmierung I**
**freiwillige Übungsaufgaben**
**14.06.2022**

---

## Aufgabe 3

Auch in dieser Woche soll wieder ein größeres Projekt umgesetzt werden: Wir wollen das Spiel Tic-Tac-Toe (auch bekannt als „Drei gewinnt“ oder „X-X-O“) implementieren (siehe https://de.wikipedia.org/wiki/Tic-Tac-Toe). Am Ende soll das Spiel auch über einen Computergegner verfügen, gegen den du antreten kannst. Das folgende Klassendiagramm gibt dir einen Überblick über das Gesamtprojekt:




Folge zur Implementierung der folgenden Anleitung:

- 🌶️
  Lege zunächst die Datei „Position.java“ an und implementiere darin die Klasse `Position`. Diese verfügt lediglich über zwei private Attribute, einen Konstruktor und Getter-Methoden.
- 🌶️
  Lege dann die Datei „GameState.java“ an. Implementiere darin zunächst das `Marker`-Enum. Ein `Marker`-Objekt ist in der Lage, den Zustand eines Feldes des Spiels zu repräsentieren (auf dem Feld befindet sich entweder ein „X“-Symbol, ein „O“-Symbol oder das Feld ist leer).
- Implementiere – ebenfalls in der Datei GameState – die Klasse `GameState`. Diese Klasse soll den aktuellen Zustand des Spielfelds repräsentieren können. Dazu enthält die Klasse ein zweidimensionales `Marker`-Array der Größe $3 \times 3$ als privates Attribut. Außerdem enthält es das Attribut `currentPlayer`, in dem der Marker des gerade aktiven Spielers bzw. Spielerin gespeichert wird. Implementiere nun die Methoden der Klasse GameState:
  - 🍌🍌
    Der Konstruktor erzeugt das zweidimensionale Array und füllt es mit dem Wert `Marker.EMPTY`. Außerdem setzt er das `currentPlayer`-Attribut auf `Marker.X`.

---

- 🌶
  Die statische Methode `getOtherPlayer()` erwartet einen Marker als Parameter. Wenn `Marker.X` übergeben wird, soll `Marker.O` zurückgegeben werden. Andernfalls wird `Marker.X` zurückgegeben.
- 🌶🌶
  Die `getCopy()`-Methode soll eine neue `GameState`-Instanz erzeugen und zurückgeben. Es soll sich dabei um eine Kopie der eigenen Instanz handeln. Es ist dafür also nötig, ein `GameState`-Objekt zu erzeugen und danach den Inhalt des gesamten Spielfelds sowie den Wert von `currentPlayer` vom aktuellen in den neuen `GameState` zu kopieren.
  *Wichtig:* Es reicht dabei nicht, die Referenz auf das `fields`-Attribut zu kopieren! Es muss jedes Array-Element einzeln kopiert werden (verschachtelte `for`-Schleifen!)
- 🌶
  Bei `getCurrentPlayer()` handelt es sich um eine Getter-Methode.
- 🌶
  Die `switchPlayers()`-Methode ändert den Wert des `currentPlayer`-Attributs (entweder von `Marker.X` auf `Marker.O` oder andersherum).
- 🌶🌶
  Die Methode `anyEmptyFields()` soll `true` zurückgeben, falls es mindestens ein freies Kästchen im Spielfeld gibt, ansonsten `false`. Unter einem „freien Kästchen“ versteht man den Wert `Marker.EMPTY`.
- 🌶🌶🌶
  Die Methode `hasPlayerWon()` stellt das Herzstück der `GameState`-Klasse dar: Die Methode soll `true` zurückgeben, wenn das übergebene Symbol (Marker.X oder `Marker.O`) gewonnen hat. Die Überprüfung muss dabei in mehreren Schritten ablaufen:
  - Gibt es eine **Zeile**, die komplett mit dem entsprechenden Symbol gefüllt ist? Falls ja, gib `true` zurück.
  - Gibt es eine **Spalte**, die komplett mit dem entsprechenden Symbol gefüllt ist? Falls ja, gib `true` zurück.
  - Ist die **aufsteigende Diagonale** komplett mit dem entsprechenden Symbol gefüllt? Falls ja, gib `true` zurück.
  - Ist die **absteigende Diagonale** komplett mit dem entsprechenden Symbol gefüllt? Falls ja, gib `true` zurück.
  - Ist keiner dieser Fälle eingetreten, gib `false` zurück.
- 🌶
  Die Methode `getDimensions()` soll die Größe des Spielfelds zurückgeben – bei uns also den Wert 3. Natürlich kannst du gerne auch deine Implementierung so flexibel gestalten, dass beliebige Spielfeldgrößen möglich sind.

---

- 🌶🌶🌶
  Die `toString()`-Methode soll überladen werden, um das Spielfeld als `String` zurückzugeben. Wenn du möchtest, kannst du auch die Zeilen- und Spaltennummern am Spielfeldrand eintragen. Der zurückgegebene `String` könnte beispielsweise so aussehen:
  ```
    012
    ---
  0|O X
  1|XXO
  2|O
  ```
  *Hinweis:* Nutze die Escape-Sequenz `\n`, um Zeilenumbrüche im String zu erzeugen.
- 🌶
  Die `isFieldEmpty()`-Methode gibt `true` zurück, falls das durch `row` und `column` angegebene Feld den Wert `Marker.EMPTY` hat.
- 🌶
  Die `setField()`-Methode setzt das durch `position` angegebene Feld auf den Wert `player`.
- 🌶
  Die private, statische Methode `markerToChar()` wandelt einen Wert vom Typ des `Marker`-Enums in einen `char` um. Ist der Wert `Marker.X`, soll der Buchstabe „X“ zurückgegeben werden. Ist der Wert `Marker.O`, soll der Buchstabe „O“ zurückgegeben werden. Bei `Marker.EMPTY` soll ein Leerzeichen zurückgegeben werden.

- Lege nun die Datei „ComputerPlayer.java“ an und implementiere dort die gleichnamige Klasse:
  - 🌶
    Der Konstruktor speichert den übergebenen Enum-Wert im Attribut `computerPlayer`. Im Attribut `humanPlayer` wird das Symbol des anderen Spielers gespeichert. Dafür kannst du die statische Methode `getOtherPlayer` der Klasse `GameState` nutzen, also:
    `humanPlayer = GameState.getOtherPlayer(computerPlayer)`
    Du siehst hier auch erstmalig, dass statische Methoden über den Klassennamen aufgerufen werden, wenn der Aufruf außerhalb der eigenen Klasse stattfindet.
  - 🌶🌶🌶
    Die Methode `getNextMove()` soll die Entscheidung treffen, auf welches Feld der Computergegner seinen nächsten Spielstein setzt. Gehe dazu folgendermaßen vor:
    - Lege dir eine Variable vom Typ `Position` mit dem Namen `possibleMove` an und initialisiere sie auf `null`.
    - Iteriere über alle Kästchen des Spielfelds und tue für jedes Kästchen folgendes:
      - Ist das Spielfeld bereits belegt, tue gar nichts.
      - Lege eine Kopie des Spielfelds an (`getCopy()`-Methode). Setze das Symbol des Computerspielers an die aktuelle Position. Überprüfe, ob dieser Zug zum Sieg führt (`hasPlayerWon()`-Methode). Falls ja, gib die aktuelle Position zurück.
      - Verwirf die Kopie des Spielfelds und lege eine weitere Kopie an. Setze das Symbol des menschlichen Spielers an die aktuelle Position.

---

Überprüfe, ob dieser Zug zur Niederlage (also zum Sieg des menschlichen Spielers) führt. Falls ja, gib die aktuelle Position zurück.
- Falls die Variable `possibleMove` eine `null`-Referenz ist, speichere in ihr die aktuelle Position.

- Gib den Wert von `possibleMove` zurück.

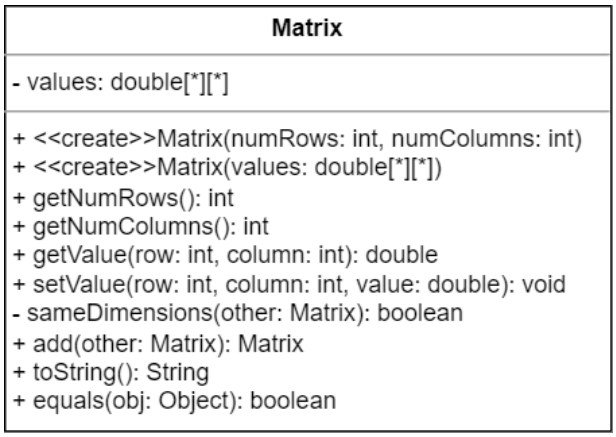
- Lege zuletzt noch die Datei „Main.java“ an und füge den Code ein, den du unter https://pastebin.com/gBLGsy6w findest. Teste damit deine Implementierung.
- Einen beispielhaften Spieldurchlauf findest du unter https://pastebin.com/KWte8QMu.

**Objektorientierte Programmierung I**
**freiwillige Übungsaufgaben**
**14.06.2022**

---

## 🍌🍌 Aufgabe 4

In dieser Aufgabe sollst du eine Klasse implementieren, die eine Matrix (im mathematischen Sinne) repräsentieren kann. Betrachte dazu zunächst das folgende Klassendiagramm:




Die Implementierungen der Konstruktoren sowie der Methoden `getNumRows()`, `getNumColumns()`, `getValue()` und `setValue()` sollten *relativ* offensichtlich sein (falls nicht, bitte nachfragen!)
Zu den restlichen Methoden:

- `sameDimensions()` soll `true` zurück geben, wenn die Matrix und die übergebene Matrix (`other`) sowohl in der Anzahl ihrer Zeilen als auch in der Anzahl ihrer Spalten übereinstimmen.
- `add()` soll die Matrix und die übergebene Matrix (`other`) addieren. Es soll dabei eine neue Matrix zurückgegeben werden. Die beiden Operanden der Operation (also die Summanden) sollen dabei nicht verändert werden.
- Überschreibe die `toString()`-Methode, sodass sie einen String zurückgibt, der die Matrix enthält. Der String könnte beispielsweise so aussehen:
  ```
  1.0, 2.0, 3.0
  4.0, 5.0, 6.0
  7.0, 8.0, 9.0
  ```
- Überschreibe die `equals()`-Methode, sodass sie `true` zurückgibt, wenn die Matrix und die übergebene Matrix (`obj`) identisch sind. Das ist genau dann der Fall, wenn sie identische Abmessungen haben und auch alle Elemente paarweise identisch sind.
  Bedenke dabei, dass du die Referenz `obj` zuerst in den Typ `Matrix` casten musst, z. B. mit `Matrix otherMatrix = (Matrix)obj;` (vgl. Aufgabe 2).

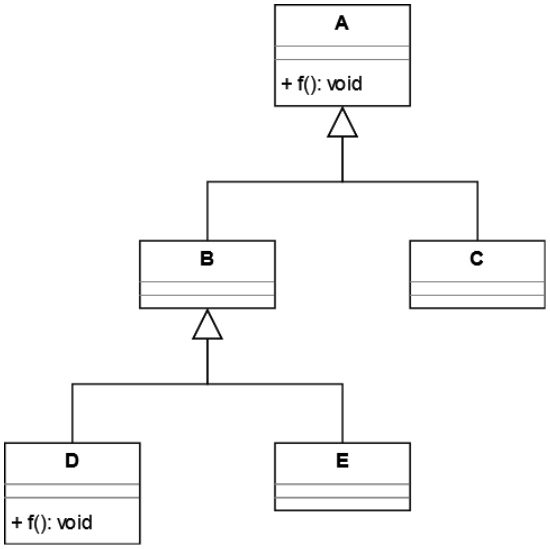
Du kannst deine Implementierung mit dem unter https://pastebin.com/CkseaVkX angegebenen Hauptprogramm testen.

**Objektorientierte Programmierung I**
**freiwillige Übungsaufgaben**
**14.06.2022**

---

## Aufgabe 5

Gegeben sei die folgende Vererbungshierarchie:




Wir nehmen an, dass wir eine Referenz vom Typ A haben, auf der wir die Methode `f()` aufrufen wollen. Die Implementierung von `f()`, die aufgerufen wird, hängt nicht vom Typ der *Referenz*, sondern vom Typ des *referenzierten Objekts* ab. Fülle die folgende Tabelle aus:

| Typ des Objekts | Welche Implementierung wird ausgeführt? |
|---|---|
| **A** | |
| **B** | |
| **C** | |
| **D** | |
| **E** | |