Prof. Dr. Jürgen Priemer: Maschinelles Lernen

Semesterbegleitende Leistung, Teil II: Bestärkendes Lernen

Abgabe der Ausarbeitungen bis zum 11.1.2026, 15 Uhr (per Mail)

1. Themenvorstellung

Beim bestärkenden Lernen (Reinforcement Learning) geht es um das „Lernen aus Erfahrung“. Anders als beim überwachten Lernen gibt es keine Trainingsdatensätze anhand derer das System trainiert werden kann, sondern das System soll anhand von Erfahrungen selbst eine Strategie erlernen, um ein möglichst gutes Ergebnis zu erzielen. Zu Beginn des Lernprozesses ist das System „dumm“ in dem Sinne, dass es nicht zwischen guten und schlechten Aktionen unterscheiden kann.

Sie sollen ein ausschließlich selbstlernendes Programm für das Spiel „Tic-Tac-Toe“ erstellen (https://de.wikipedia.org/wiki/Tic-Tac-Toe). **Ihr Programm soll möglichst mit einem neuronalen Netz arbeiten. Die explizite Verwendung von Spielbäumen und/oder Strategieregeln ist nicht erlaubt.** Ihr Programm weiß vor Beginn des Lernprozesses nur,

- welche Züge erlaubt sind (sehr einfach: Nur auf freie Felder)
- wann ein Spiel beendet ist und wie es gewertet wird (Sieg für X, Sieg für O, unentschieden)

Der Lernprozess kann durch Spiel des Programms gegen sich selbst oder gegen einen festen Gegner erfolgen. Das Programm muss in der Lage sein, sowohl mit „X“ als auch mit „O“ spielen zu können.

2. Programmentwicklung

Das Programm soll in Java entwickelt werden. Als Ausgangspunkt verwenden Sie bitte die JAR-Datei **`TicTacToe.jar`**, die einige wesentliche Klassen des Programms enthält.

Die Benutzung von Frameworks (z.B. für neuronale Netze) ist erlaubt, solange die Logik für den Lernprozess in Ihrem eigenen Programmcode steckt.

Die Verwendung von Spielbäumen, d.h. die Verwendung des Minimax- oder Alpha-Beta-Verfahrens ist ausdrücklich verboten.

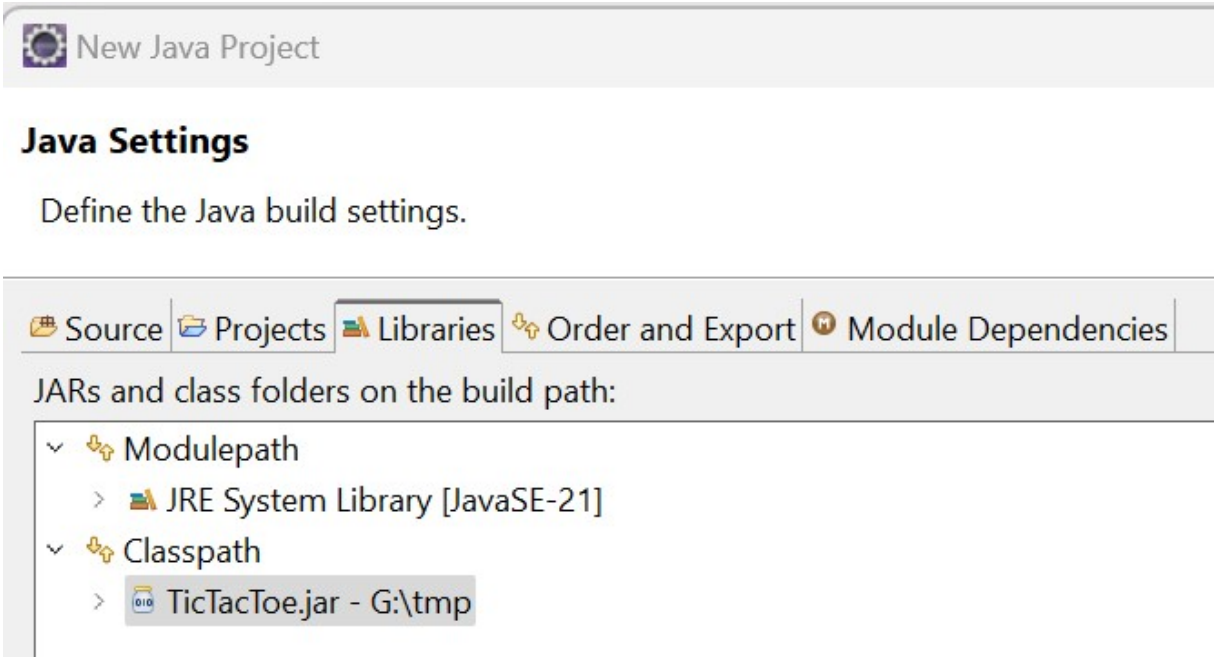
3. Anlegen eines Projekts; Erstellung von Spielern

Ihr Spieler muss in der Programmiersprache Java[1] erstellt werden und die vorgegebene Java-Bibliothek TicTacToe.jar nutzen.

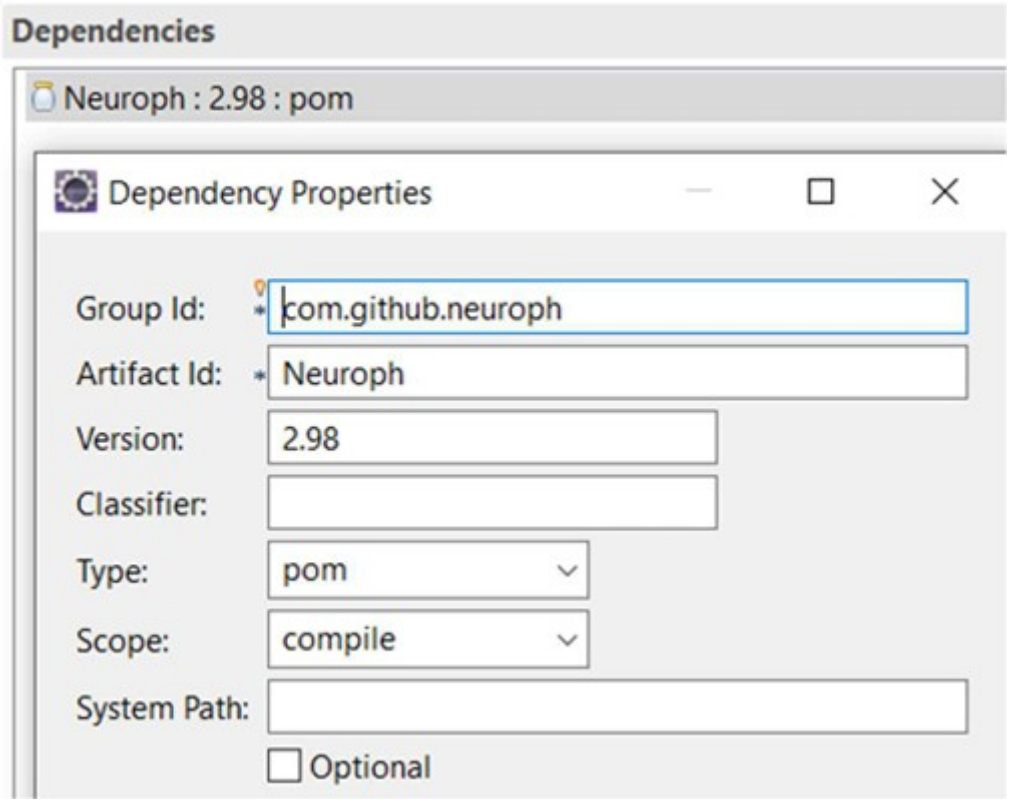
1. Um einen eigenen Spieler zu implementieren, müssen Sie zunächst einige bereits vorhandene Java-Klassen und Schnittstellen vom Moodle-Server herunterladen: Wählen Sie die Datei TicTacToe.jar und laden Sie diese herunter.

[1] Ich verwende hier die JDK-Version 21 (Long Time Support). Das Projekt sollte aber auch mir einer neueren JDK-Version lauffähig sein.

2. Erstellen Sie jetzt in Eclipse ein Java-Projekt (z.B. TicTacToeGruppe1). Fügen Sie dabei die gerade heruntergeladene JAR-Datei als Library hinzu (zweites Fenster bei der Projekterstellung, „Add external Jars“):

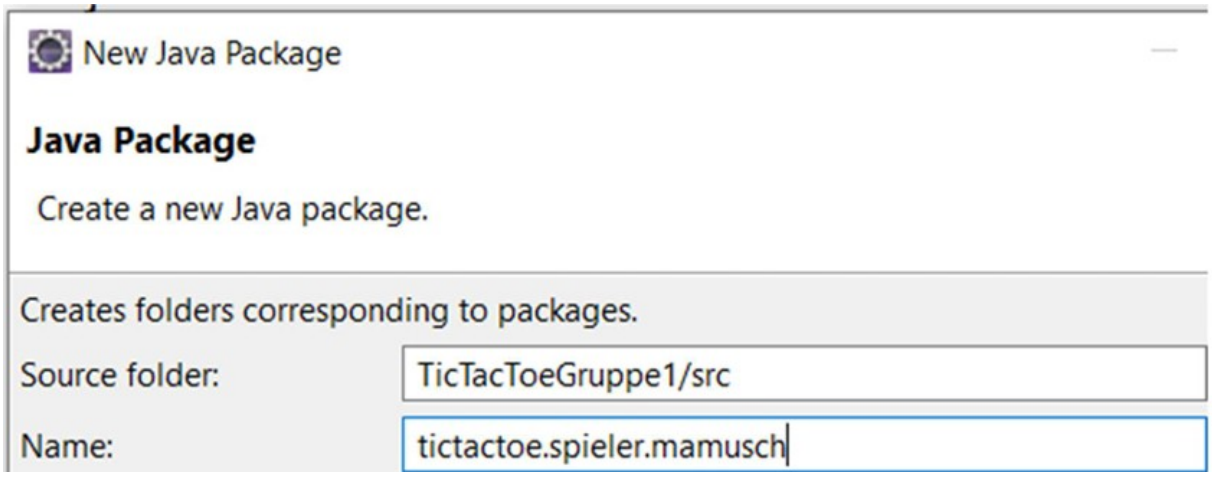


3. Wenn Sie ein Framework für neuronale Netze, wie z.B. Neuroph, verwenden möchten, können Sie dieses jetzt ebenfalls einbinden. Sie können entweder die JAR-Datei für Ihr Framework herunterladen (für Neuroph z.B. https://sourceforge.net/projects/neuroph/files/neuroph-2.98/neuroph-2.98.zip/download) und dann ebenfalls wie in Schritt 2 einbinden oder Sie wandeln Ihr Projekt in ein Maven-Projekt um und nehmen die Einbindung über Maven vor.

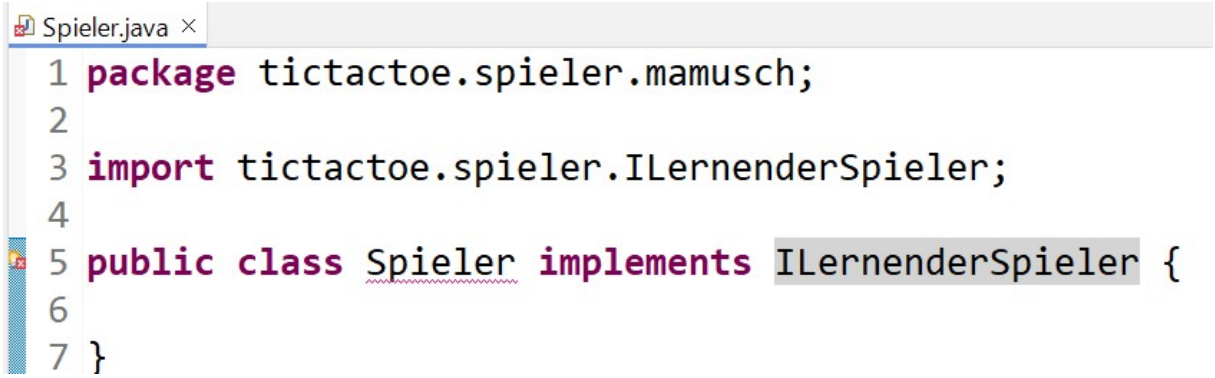


4. Jetzt können Sie die Java-Klasse für Ihren TicTacToe-Spieler anlegen. Dafür müssen Sie zunächst ein Paket tictactoe.spieler und dort ein Unterpaket für Ihre Gruppe anlegen, auf das wir nachfolgend eingehen. Wählen Sie als Namen für das Unterpaket eine im gesamten Kurs eindeutige Bezeichnung; am besten verwenden Sie Ihren Gruppennamen oder ein entsprechendes Kürzel.

   Beispiel: Die Gruppe, die aus den Studenten Maier, Mueller und Schmidt besteht, wählt als Gruppenkürzel mamusch. Sie legt ein Unterpaket mamusch in tictactoe.spieler an (also tictactoe.spieler.mamusch).



5. In diesem Paket implementieren Sie Ihren TicTacToe-Spieler. Ihre Klasse muss zwingend **Spieler** heißen und die Schnittstelle **ILernenderSpieler** implementieren. Diese stammt aus dem Paket **tictactoe.spieler** aus der JAR-Datei.



   Um den Fehler in Zeile 5 zu beseitigen, müssen Sie die von der Schnittstelle ILernenderSpieler verlangten Methoden implementieren:

Spieler.java

```java
package tictactoe.spieler.mamusch;

import java.io.IOException;

import tictactoe.Farbe;
import tictactoe.IllegalerZugException;
import tictactoe.Zug;
import tictactoe.spieler.IAbbruchbedingung;
import tictactoe.spieler.ILernenderSpieler;

public class Spieler implements ILernenderSpieler {

    @Override
    public Zug berechneZug(Zug arg0, long arg1, long arg2) throws IllegalerZugException {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public Farbe getFarbe() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public String getName() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public void neuesSpiel(Farbe arg0, int arg1) {
        // TODO Auto-generated method stub

    }

    @Override
    public void setFarbe(Farbe arg0) {
        // TODO Auto-generated method stub

    }

    @Override
    public void setName(String arg0) {
        // TODO Auto-generated method stub

    }

    @Override
    public void ladeWissen(String arg0) throws IOException {
        // TODO Auto-generated method stub

    }

    @Override
    public void speichereWissen(String arg0) throws IOException {
        // TODO Auto-generated method stub

    }

    @Override
    public boolean trainieren(IAbbruchbedingung arg0) {
        // TODO Auto-generated method stub
        return false;
    }

}
```

Abbildung 1: Template für einen lernenden Spieler

6. Die Zeitlimitierung müssen Sie bei der Implementierung von **`neuesSpiel(….)`** noch nicht berücksichtigen, da diese im Rahmenprogramm noch nicht implementiert wurde. Bei **`berechneZug(….)`** erhalten Sie auch noch keine Angaben über die bisher verbrauchte Rechenzeit.

7. Damit Sie Ihren Spieler trainieren und testen können, brauchen Sie jetzt noch eine Umgebung in der dies stattfinden kann. Diese finden Sie ebenfalls auf Moodle in Form der Klasse Wettkampf.java. Damit Sie Ihre eigenen Tests durchführen können, erhalten Sie die Klasse im Quelltext.

   Bitte beachten Sie, dass der spieler2 in Zeile 17 nur als Platzhalter für Ihren Spieler steht. Die Klasse ReinforcementSpielerLeer ist nur ein nicht-funktionierendes Template für einen lernenden Spieler.

```java
package tictactoe;


import tictactoe.spieler.*;
import tictactoe.spieler.beispiel.ReinforcementSpielerLeer;
import tictactoe.spieler.beispiel.Zufallsspieler;

/** Klasse um einen Wettkampf zwischen zwei Spielern durchzuführen */
public class Wettkampf {

    public static void main(String[] args) {
        ISpieler spieler1 = new Zufallsspieler("Zufall");

        //Fügen Sie nachfolgend Ihren eigenen lernenden Spieler ein.
        //Dieser muss die Schnittstelle ILernenderSpieler implementieren
        //Der hier nachfolgende Spieler ist nur ein (nicht funktionierendes) Muster
        ILernenderSpieler spieler2 = new ReinforcementSpielerLeer("Lernender Spieler");

        TicTacToe spiel = new TicTacToe();
        ISpieler gewinner;
        int gewinne1=0;
        int gewinne2=0;

        System.out.println("Vor dem Training");
        System.out.println(spieler1.getName() + " gegen " + spieler2.getName());
        System.out.println("=======================================================");
        for (long i=0; i<1000; i++) {
            gewinner = spiel.neuesSpiel(spieler1,spieler2,150,false);
            if (gewinner==spieler1) {
                gewinne1++;
            }
            else {
                if (gewinner==spieler2)
                gewinne2++;
            }
            gewinner = spiel.neuesSpiel(spieler2,spieler1,150,false);
            if (gewinner==spieler1) {
                gewinne1++;
            }
            else {
                if (gewinner==spieler2)
                gewinne2++;
            }

        }
        System.out.println("Gewinne " + spieler1.getName() + ": " + gewinne1 + ". Gewinne " + spieler2.getName() + ": "+ gewinne2);
        System.out.println();

        gewinne1=0;
        gewinne2=0;
        //Hier würde jetzt das Training kommen!
        System.out.println("Starte Training mit 200000 Iterationen. Bitte haben Sie etwas Geduld!");
        long starttime = System.currentTimeMillis();
        spieler2.trainieren(new AbbruchNachIterationen(200000));
        long endtime = System.currentTimeMillis();
        System.out.println("Training beendet. Gesamtdauer in Sekunden: " + ((endtime - starttime) / 1000));
        System.out.println(spieler1.getName() + " gegen " + spieler2.getName());
        System.out.println("=======================================================");
        for (long i=0; i<1000; i++) {
            gewinner = spiel.neuesSpiel(spieler1,spieler2,150,false);
            if (gewinner==spieler1) {
                gewinne1++;
            }
            else {
                if (gewinner==spieler2)
                gewinne2++;
            }
        }
        System.out.println("Gewinne " + spieler1.getName() + ": " + gewinne1 + ". Gewinne " + spieler2.getName() + ": "+ gewinne2);
        System.out.println();
        System.out.println("Ein Einzelspiel im DEBUG-Modus, lernender Spieler startet mit X");
        System.out.println("============================================================");
        System.out.println("Gewonnen hat: " + spiel.neuesSpiel(spieler2, spieler1,150,true).getName());
        System.out.println();
        System.out.println("Ein Einzelspiel im DEBUG-Modus, lernender Spieler zweiter mit O");
        System.out.println("============================================================");
        System.out.println("Gewonnen hat: " + spiel.neuesSpiel(spieler1, spieler2,150,true).getName());

    }
}
```

Abbildung 2: Die Klasse Wettkampf

4. Weitere Klassen und Schnittstellen (Auszug Javadoc)

Schnittstelle ISpieler

Package tictactoe.spieler

Schnittstelle ISpieler

Alle bekannten Unterschnittstellen:
`ILernenderSpieler`

Alle bekannten Implementierungsklassen:
`ReinforcementSpieler`, `ReinforcementSpielerLeer`, `Zufallsspieler`

`public interface ISpieler`

Schnittstelle für alle ISpieler.

Autor:
Jürgen Priemer

Methodenübersicht

Alle Methoden | Instanzmethoden | Abstrakte Methoden

Modifizierer und Typ	Methode	Beschreibung
`Zug`	`berechneZug(Zug vorherigerZug, long zeitKreuz, long zeitKreis)`	Diese Methode muss zur Zugberechnung implementiert werden
`Farbe`	`getFarbe()`	
`String`	`getName()`	
`void`	`neuesSpiel(Farbe meineFarbe, int bedenkzeitInSekunden)`	Ein neues Spiel wird gestartet.
`void`	`setFarbe(Farbe farbe)`	Eigene Farbe setzen.
`void`	`setName(String name)`	

Methodendetails

neuesSpiel

```
void neuesSpiel(Farbe meineFarbe,
                int bedenkzeitInSekunden)
```

Ein neues Spiel wird gestartet. Der ISpieler muss schon vorher trainiert worden sein!

Parameter:
`meineFarbe` -
`bedenkzeitInSekunden` -

getName

```
String getName()
```

Gibt zurück:
Bezeichnung des eigenen Spielers

setName

```
void setName(String name)
```

Parameter:
`name` - Name des Spielers

getFarbe

```
Farbe getFarbe()
```

Gibt zurück:
eigene Farbe

setFarbe

```
void setFarbe(Farbe farbe)
```

Eigene Farbe setzen.

Parameter:
`farbe` -

berechneZug

```
Zug berechneZug(Zug vorherigerZug,
                long zeitKreuz,
                long zeitKreis)
         throws IllegalerZugException
```

Diese Methode muss zur Zugberechnung implementiert werden

Parameter:
`vorherigerZug` - : Der letzte Zug des Gegners. null falls der Gegner vorher keinen Zug gemacht hat
`zeitKreuz` - : Bisher verbrauchte Zeit des Spielers mit Farbe Kreis in Millisekunden
`zeitKreis` - : Bisher verbrauchte Zeit des Spielers mit Farbe Kreuz in Millisekunden

Gibt zurück:
berechneten Zug

Löst aus:
`IllegalerZugException` - wenn vorherigerZug illegal ist (nach Meinung des Spielers)

Schnittstelle ILernenderSpieler

Package tictactoe.spieler

Schnittstelle ILernenderSpieler

Alle Superschnittstellen:

ISpieler

Alle bekannten Implementierungsklassen:

ReinforcementSpieler, ReinforcementSpielerLeer

```
public interface ILernenderSpieler
extends ISpieler
```

Schnittstelle für alle lernenden Spieler

Methodenübersicht

Alle Methoden | Instanzmethoden | Abstrakte Methoden

Modifizierer und Typ	Methode	Beschreibung
void	**ladeWissen**(String name)	
void	**speichereWissen**(String name)	
boolean	**trainieren**(**IAbbruchbedingung** abbruchbedingung)	Training für einen lernenden Spieler.

Von Schnittstelle geerbte Methoden tictactoe.spieler.ISpieler

berechneZug, getFarbe, getName, neuesSpiel, setFarbe, setName

Methodendetails

trainieren

```
boolean trainieren(IAbbruchbedingung abbruchbedingung)
```

Training für einen lernenden Spieler. Die Abbruchbedingung regelt, wie lange gelernt wird. Wenn das Training erfolgreich war, wird true zurückgeliefert.

Parameter:

abbruchbedingung -

Gibt zurück:

speichereWissen

```
void speichereWissen(String name)
             throws IOException
```

Löst aus:

IOException

ladeWissen

```
void ladeWissen(String name)
         throws IOException
```

Löst aus:

IOException

Schnittstelle IAbbruchbedingung

Package tictactoe.spieler

Schnittstelle IAbbruchbedingung

Alle bekannten Implementierungsklassen:

AbbruchNachIterationen, AbbruchNachZeit

```
public interface IAbbruchbedingung
```

Methodenübersicht

Alle Methoden | **Instanzmethoden** | **Abstrakte Methoden**

Modifizierer und Typ	Methode	Beschreibung
boolean	abbruch()	Liefert true, wenn das Training abgebrochen werden soll.

Klasse Zug

Package tictactoe

Klasse Zug

java.lang.Object
  tictactoe.Zug

```
public class Zug
extends Object
```

Konstruktorübersicht

Konstruktoren

Konstruktor	Beschreibung
Zug(int zeile, int spalte)	Ein Zug besteht aus der Angabe einer Zeile und einer Spalte.

Methodenübersicht

Alle Methoden | **Instanzmethoden** | **Konkrete Methoden**

Modifizierer und Typ	Methode	Beschreibung
int	getSpalte()	
int	getZeile()	

Von Klasse geerbte Methoden java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Konstruktordetails

Zug

```
public Zug(int zeile,
           int spalte)
```

Ein Zug besteht aus der Angabe einer Zeile und einer Spalte. Die oberste Zeile hat die Nummer 0. Die linke Spalte hat die Nummer 0.

Parameter:

zeile - Zeile des Zugs

spalte - Spalte des Zugs

Klasse Farbe

Package tictactoe

Enum-Klasse Farbe

java.lang.Object
    java.lang.Enum<Farbe>
        tictactoe.Farbe

Alle implementierten Schnittstellen:

`Serializable`, `Comparable<Farbe>`, `Constable`

```
public enum Farbe
extends Enum<Farbe>
```

Farbe: Kreuz, Kreis oder Leer

Verschachtelte Klassen - Übersicht

Von Klasse geerbte verschachtelte Klassen/Schnittstellen java.lang.Enum

`Enum.EnumDesc<E extends Enum<E>>`

Enum-Konstanten - Übersicht

Enum-Konstanten

Enum-Konstante	Beschreibung
`Kreis`	
`Kreuz`	
`Leer`	

Methodenübersicht

Alle Methoden | **Statische Methoden** | **Instanzmethoden** | **Konkrete Methoden**

Modifizierer und Typ	Methode	Beschreibung
`Farbe`	`opposite()`	Gegenteil einer Farbe: Kreuz ==> Kreis
`int`	`toInput()`	1 für Kreis, -1 für Kreuz, 0 wenn leer
`int`	`toInt()`	1 für Kreis, 2 für Kreuz, 0 wenn leer
`String`	`toString()`	Gibt O, X oder _ aus
`static Farbe`	`valueOf(String name)`	Gibt die Enum-Konstante dieser Klasse mit dem angegebenen Namen zurück.
`static Farbe[]`	`values()`	Gibt ein Array mit den Konstanten dieser Enum-Klasse in der Reihenfolge ihrer Deklaration zurück.

Klasse TicTacToe

```
package tictactoe;


import tictactoe.spieler.Farbe;

/** Definiert das Spielfeld und enthält Methoden, um ein Spiel durchzuführen.
 *
 * @author Jürgen Priemer
 *
 */
public class TicTacToe {

    private Spielfeld feld = new Spielfeld();

    private void neuesSpiel() {

    /**
     * Erzeugt ein neues Spiel, das bis zum Ende durchgespielt wird.
     * Zur Zugberechnung wird berechneZug(..) für den jeweiligen Spieler aufgerufen.
     * @param spielerKreis (spielt Kreis)
     * @param spielerKreuz (spielt Kreuz)
     * @param bedenkzeitInSek (Gesamtbedenkzeit für jeden Spieler)
     * @param debug Wenn True wird das Spielfeld auf der Konsole ausgegeben
     * @return Spieler gewinner
     */
    public Spieler neuesSpiel(Spieler spielerKreis, Spieler spielerKreuz, int bedenkzeitInSek, boolean debug) {

    /**
     * Durchführen eines Zuges für den Spieler mit der angegebenen Farbe
     * @param spieler: Farbe des Spielers am Zug
     * @param zug
     * @return aktueller Spielstand
     * @throws IllegalerZugException
     */
    private Spielstand machZug(Farbe spieler, Zug zug) throws IllegalerZugException {

     * Statische Methode: Prüft für ein Spielfeld, ob der Spieler mit der angegebenen Farbe gewonnen hat.
    private static Spielstand pruefeGewinn(Spielfeld feld, Farbe spieler) {

    private static Spielstand spielende(Farbe f1, Farbe f2) {

    public void printFeld() {

    /** Nur für Testzwecke */
    public static void main(String[] args) {

}
```

5. Ablauf des Wettkampfs bzw. eines Spiels

1. Erzeugen der beteiligten Spieler durch **Aufruf des jeweiligen Konstruktors**:
   Einziger zwingender Parameter ist der Name des Spielers (als String). Weitere Parameter sind optional möglich.
   Es wird ein neuer Spieler erzeugt. Der Spieler ist selbst für die Verwaltung eines Spielfelds und das Nachhalten der durchgeführten Züge verantwortlich. Am Ende der Initialisierung muss der Spieler trainiert sein; am besten ist es aber, den Spieler schon vorher unabhängig vom Wettkampf zu trainieren und das erzeugte neuronale Netz abzuspeichern.

2. Start eines neuen Spiels, an dem der Spieler beteiligt ist durch Aufruf von **`neuesSpiel(farbe, bedenkzeit)`**. Der Spieler muss sich die mitgeteilte Farbe merken und sein internes Spielfeld leeren. Der Parameter **`bedenkzeit`** gibt die insgesamt für das Spiel verfügbare Bedenkzeit in Sekunden an. ***Dieser Parameter wird im Moment noch nicht genutzt.***
   Der Spielstart selbst wird initiiert durch die Methode **`neuesSpiel(spielerKreis, spielerKreuz, bedenkzeitInSekunden)`** der Klasse **`TicTacToe`** aus dem Paket **`tictactoe`**.

3. Aufforderung zur Berechnung eines Spielzugs durch Aufruf von **`berechneZug(vorherigerZug, zeitKreis, zeitKreuz)`**. Im Parameter **`vorherigerZug`** ist der letzte Zug des Gegners enthalten. Der Spieler muss diesen Zug im eigenen Spielfeld vermerken, damit er weiß, dass das entsprechende Feld besetzt ist. Führt man selbst den ersten Zug in einem neuen Spiel durch, ist der Parameter **`vorherigerZug`** **`null`**. Die Parameter **`zeitKreis`** und **`zeitKreuz`** geben die bisher verbrauchte Zeit der entsprechenden Farbe in Millisekunden an (***diese Parameter enthalten im Moment noch keine Werte***).
   Als Ergebnis der Zugberechnung muss der berechnete Zug zurückgegeben werden.

4. Schritt 3 läuft ab, solange Züge möglich sind und noch kein Spieler gewonnen hat. Andernfalls endet das Spiel. Als Ergebnis des Aufrufs **`neuesSpiel(…)`** aus **`TicTacToe`** wird der Gewinner (als **`Spieler`**) zurückgegeben. Bei einem Unentschieden wird **`null`** zurückgeliefert.

6. Beispiel für die Implementierung eines Spielers

Das nachfolgende Beispiel zeigt die Implementierung eines nicht-lernenden Spielers, der zufällige (erlaubte) Züge macht. Sie können diesen Testspieler auch als Gegenspieler für Ihren eigenen Spieler nutzen.

Der Spieler wird im Unterpaket **`tictactoe.spieler.beispiel`** implementiert:

```java
package tictactoe.spieler.beispiel;

import java.util.Random;

import tictactoe.Farbe;
import tictactoe.Spielfeld;
import tictactoe.Zug;
import tictactoe.spieler.ISpieler;

/**
 * Eine Musterimplementierung, die zufällige Züge macht.
 * @author Jürgen Priemer
 *
 */
public class Zufallsspieler implements ISpieler {
    private Random zufall = new Random();
    private Spielfeld spielfeld;
    private String meinName;
    private Farbe meineFarbe;

    public Zufallsspieler(String name) {
        this.meinName = name;
    }

    @Override
    public void neuesSpiel(Farbe meineFarbe, int bedenkzeitInSekunden) {
        this.meineFarbe = meineFarbe;
        spielfeld = new Spielfeld();
    }

    @Override
    public void setName(String name) {
        this.meinName = name;
    }

    @Override
    public String getName() {
        return meinName;
    }

    @Override
    public void setFarbe(Farbe farbe) {
        meineFarbe = farbe;

    }

    @Override
    public Farbe getFarbe() {
        return meineFarbe;
    }

    @Override
    public Zug berechneZug(Zug vorherigerZug, long zeitKreis, long zeitKreuz) {
        //Zunächst eventuellen Zug des Gegners durchführen
        if (vorherigerZug != null)
            spielfeld.setFarbe(vorherigerZug.getZeile(),
                               vorherigerZug.getSpalte(),
                               meineFarbe.opposite());
        Zug neuerZug;
        do {
            neuerZug = new Zug(zufall.nextInt(3), zufall.nextInt(3));
        }
        while (spielfeld.getFarbe(neuerZug.getZeile(),neuerZug.getSpalte()) != Farbe.Leer);
        spielfeld.setFarbe(neuerZug.getZeile(), neuerZug.getSpalte(), meineFarbe);
        return neuerZug;
    }
}
```

7. Abgaben

Die **Abgabe** umfasst die beiden folgenden Bestandteile:

- eine Ausarbeitung[2] (Umfang ca. 5 bis 10 Seiten) mit (mindestens) dem folgenden Inhalt
    - Beschreibung des Problems (Versuchen Sie hier auch eine grobe Abschätzung, wie viele unterschiedliche Spielsituationen möglich sind)
    - Kurzer allgemeiner Theorieteil über Reinforcement Learning (RL). Nehmen Sie anschließend eine Fokussierung auf das von Ihnen bearbeitete Problem vor. Gehen Sie auf den Einsatz neuronaler Netze im Zusammenhang mit RL ein. Beschreiben Sie, warum bzw. wann der Einsatz von neuronalen Netzen im Zusammenhang mit RL sinnvoll ist.
    - Beschreiben Sie Ihre konkrete Lösung so verständlich, dass sie einer Ihrer Kommilitonen aus einer anderen Gruppe verstehen kann. Versuchen Sie zu Beginn der Beschreibung einen grafischen Überblick (z.B. in Form eines Klassendiagramms, wenn Sie objektorientiert vorgehen) zu geben. Erläutern Sie die Bestandteile Ihres Codes, die im Zusammenhang mit dem Lernprozess relevant sind. Weisen Sie anhand eines Beispiels nach, dass Ihr Programm tatsächlich lernt (z.B. Anzahl der Siege gegen einen Gegner ist nach dem Lernen signifikant höher als vorher).
    - Geben Sie einen abschließenden Überblick und zählen Sie festgestellte Probleme und mögliche Lösungen bzw. Erweiterungen auf.
- Ein Eclipse-Projekt, das Ihr gesamtes Programm im Quelltext enthält mit einer kurzen Anleitung, wie das Projekt genutzt werden kann. Bei Unklarheiten wenden Sie sich bitte an den Professor.

[2] Wie in jeder wissenschaftlichen Arbeit dürfen Sie auf Quellen zurückgreifen. Diese Quellen dürfen auch beschreiben, wie man Tic-Tac-Toe mithilfe von bestärkendem Lernen spielt. Wichtig ist aber, dass Sie angeben, welche Quellen Sie benutzt haben.