Einführung in die Informatik

Jochen Hoenicke

Software Engineering
Albert-Ludwigs-University Freiburg

Sommersemester 2014

Fallstricke

- Java unterstützt keine Laufzeittypen für Generics. Java merkt sich zur Laufzeit nur den Basistyp List, nicht die generischen Parameter.
- Eine Typkonvertierung z.B. auf `List<String>` kann nicht zur Laufzeit überprüft werden.
- Auch `instanceof` funktioniert nicht auf generischen Typen.
- Eine Typkonvertierung auf `E` in der Klasse $List < E >$ kann zur Laufzeit nicht überprüft werden.
- Man kann kein Array vom Typ `E[]` erzeugen.

Kompatibilität

Aus historischen Gründen gibt es in Java zu der Klasse `List<E>` den Typ `List`, der sich wie `List<Object>` verhält. Auch hier gibt es Compilerwarnungen.

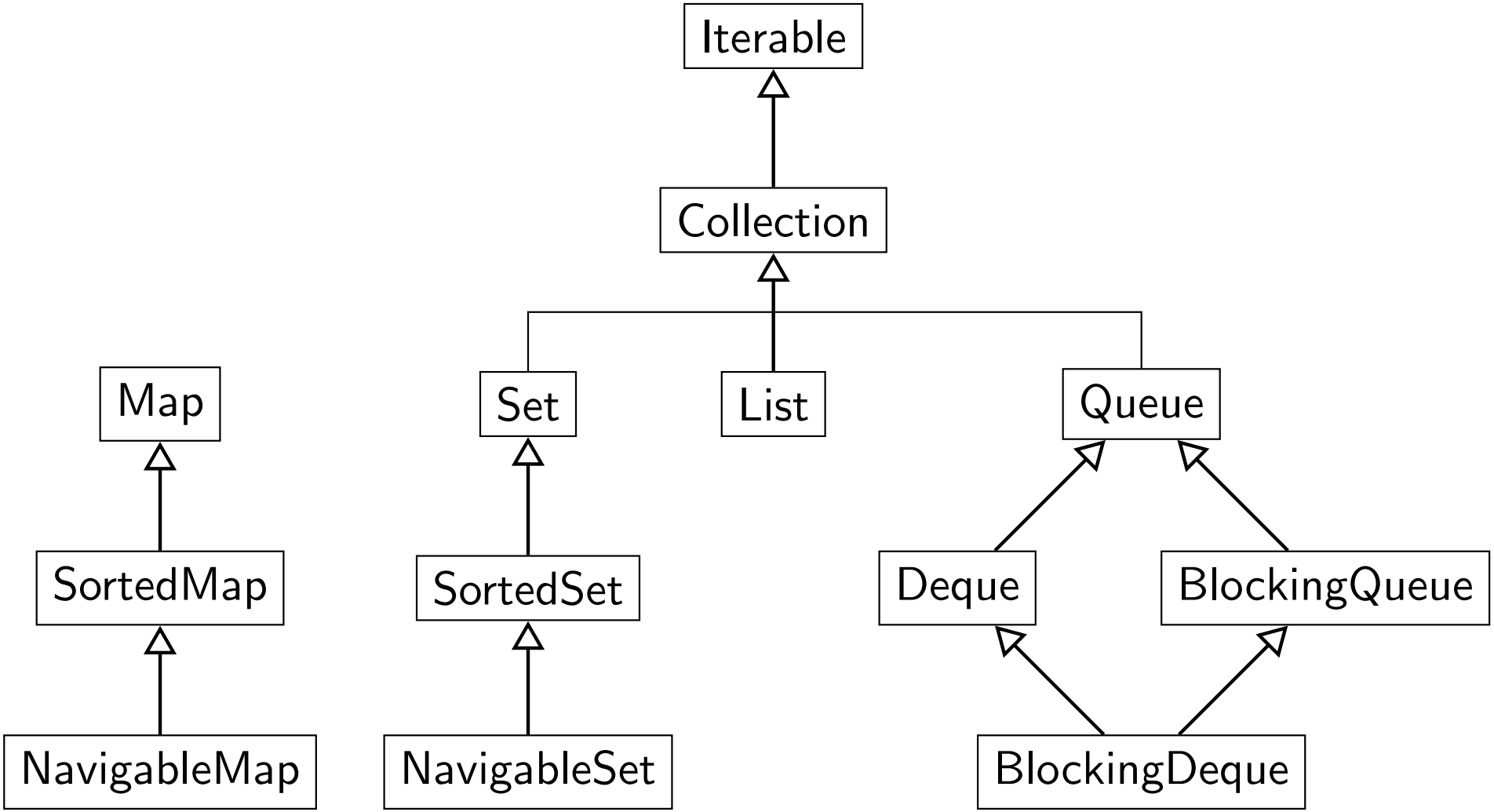
Collection-Bibliothek

Collection-Bibliothek

Die Java-Bibliothek stellt eine Reihe von Klassen zur Verfügung um Daten zu speichern.

- Container-Klassen um Objekte in verketteten Listen, Arrays, oder Bäumen zu speichern.
- Abbildungen (Maps), um Objekte miteinander zu verknüpfen
- Interfaces, die es erlauben, Objekte hinzuzufügen, entfernen, suchen, iterieren.
- Verschiedene Implementierungen, die für unterschiedliche Zwecke geeignet sind.

Collection-Hierarchie: Interfaces



Iterable

Das Interface `Iterable<E>` hat nur eine Methoden:

- `Iterator<E> iterator()` gibt einen Iterator zurück der alle Objekte aufzählt.

Der Iterator hat wiederum drei Methoden:

- `boolean hasNext()`: gibt true zurück, wenn es noch weitere Objekte gibt.
- `E next()`: gibt bei jedem Aufruf das nächste Element zurück. Wirft eine Exception, wenn es kein weiteres Element gibt.
- `void remove()` (optional): löscht das letzte Element aus der Collection.

Damit kann man alle Objekte aufzählen.

Iterable und for-Schleifen

Wenn eine Klasse `Iterable<E>` implementiert gibt es eine „schöne“ Syntax der for-Schleife:

```
Iterable<String> list = ....;
for (String s : liste) {
    ....
}
```

Hier wird der Schleifenrumpf für jedes Element der Liste einmal aufgerufen. Es ist eine Abkürzung für:

```
Iterable<String> list = ....;
Iterator<String> iterator = liste.iterator();
while (iterator.hasNext()) {
    String s = iterator.next();
    ....
}
```

Collection

Das Interface `Collection<E>` hat zusätzliche Methoden:

- `boolean add(E e)` fügt ein Element hinzu.
- `boolean addAll(Collection<? extends E> c)` fügt alle Elemente einer anderen Collection hinzu. r anderen Collection.
- `boolean clear()` löscht alle Elemente.
- `boolean contains(Object o)` prüft ob ein Element enthalten ist.
- `boolean containsAll(Collection<?> e)` prüft ob alle Elemente enthalten sind.
- `boolean isEmpty()` prüft ob die Collection leer ist.
- `boolean remove(Object o)` löscht ein Element.
- `boolean removeAll(Collection<?> c)` löscht alle Elemente eine
- `boolean retainAll(Collection<?> c)` löscht alle Elemente, die nicht in der anderen Collection liegen.
- `int size()` gibt die Anzahl der Elemente zurück.
- `T[] toArray(T[])` gibt ein Array mit allen Elementen zurück.

equals und hashCode

Die Klasse `Object` definiert zwei für Collections relevante Methoden:

- `boolean equals(Object o)` vergleicht das Objekt mit einem anderen Objekt, und gibt `true` zurück, falls die Objekte gleich sind.

  Eigenschaften `o.equals(o)== true`, `o.equals(null)== false` und `o1.equals(o2)== o2.equals(o1)` und Transitivität sollten gelten.
- `int hashCode()` gibt einen „Fingerabdruck“ des Objekts als `int` zurück. Gleiche Objekte müssen den gleichen Fingerabdruck haben. Verschiedene Objekte sollten (aus Effizienzgründen) verschiedene Fingerabdrücke haben.

  Letzteres ist nicht immer möglich, da es nicht genug Integer gibt. Es sollte aber mit großer Wahrscheinlichkeit gelten.

Collections: Übersicht

Es gibt vier Hauptarten von Collections

- `Set`: Speichert Objekte ohne Duplikate in beliebiger Reihenfolge (oder sortiert in `SortedSet`).
- `List`: Speichert Objekte mit Duplikaten in fester Reihenfolge.
- `Queue`: Speichert Objekte mit Duplikaten (mit oder ohne Reihenfolge) mit schnellem Zugriff auf das erste Element.
  Die Erweiterung `Deque` (double ended queue) erlaubt das schnelle hinzufügen und entfernen sowohl vorne, als auch hinten.
- `Map`: Speichert Assoziation zwischen Schlüssel- und Wertobjekten, mit schnellem Zugriff auf einen Wert anhand des Schlüssels. Obwohl `Map` selbst keine Collection ist, bilden die Schlüssel ein `Set` und die Werte eine `Collection`.

Implementierungen

Zu jedem Interface in der Collection-Hierarchie gibt es ein oder mehrere Implementierungen:

- `Map`: `HashMap` und `LinkedHashMap`
- `SortedMap`/`NavigableMap`: `TreeMap`
- `Set`: `HashSet` und `LinkedHashSet`
- `SortedSet`/`NavigableSet`: `TreeSet`
- `List`: `ArrayList` und `LinkedList`
- `Queue`: `PriorityQueue`.
- `Deque`: `ArrayDeque`, `LinkedList`

Sie unterscheiden sich vor allem in der Effizienz einiger Methoden.

Die Interfaces `Comparator<E>` und `Comparable<E>`

Für `SortedSet` muss man Objekte vergleichen können. Dafür gibt es das Interface `Comparator<E>`, mit einer Methode:

- `int compare(E x, E y)`: Gibt einen negativen Wert zurück, falls $x < y$, Null, wenn x = y und positiven Wert wenn $x > y$ ist. Dabei kann größer/kleiner eine beliebige totale Ordnung sein.

Statt einen `Comparator` kann eine Klasse auch eine natürliche Ordnung auf sich selbst definieren und `Comparable<E>` implementieren:

- `int compareTo(E y)` hat den gleichen Kontrakt wie $\mathit{compare}(\mathit{this}, y)$.

Teil IX

Alapo – Design eines größeren Programms

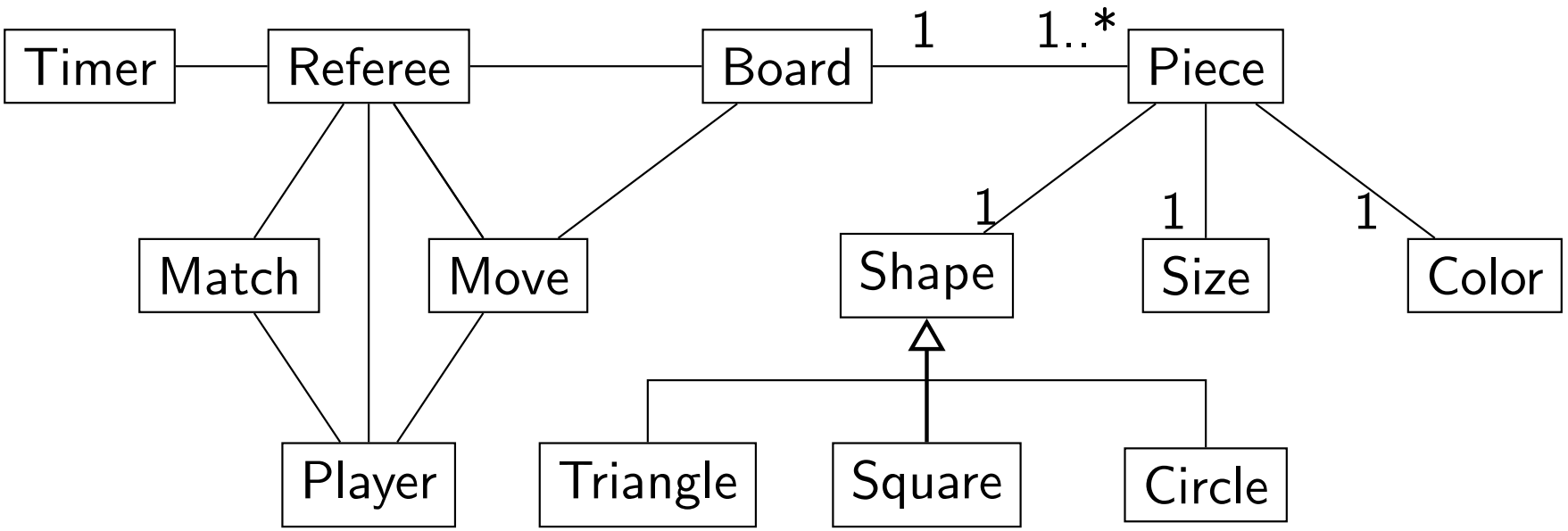
Alapo

Als Programmierübung wollen wir ein Programm schreiben, dass Alapo spielt. Diese Programme sollen am Ende in einem Turnier antreten.

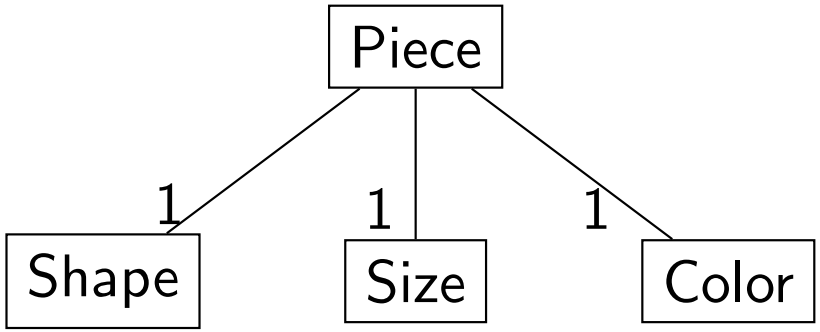
In einer Alapo-*Partie* treten zwei *Spieler* gegeneinander an. Sie berechnen abwechselnd *Züge*, die auf einem *Brett* ausgeführt werden. Auf dem Brett befinden sich *Spielfiguren* von verschiedenen *Formen* (*Dreieck*, *Quadrat*, *Kreis*), *Größen* (klein, groß) und *Farben* (schwarz, weiß). Ein *Schiedsrichter* überprüft diese Züge. Weiter solle der Schiedsrichter die *Bedenkzeit* der Spieler zu überwachen.

Klassen

In der objektorientierten Programmierung werden als erstes die benötigten Klassen aufgeschrieben. Die Hauptwörter der Beschreibung geben einen ersten Anhaltspunkt:



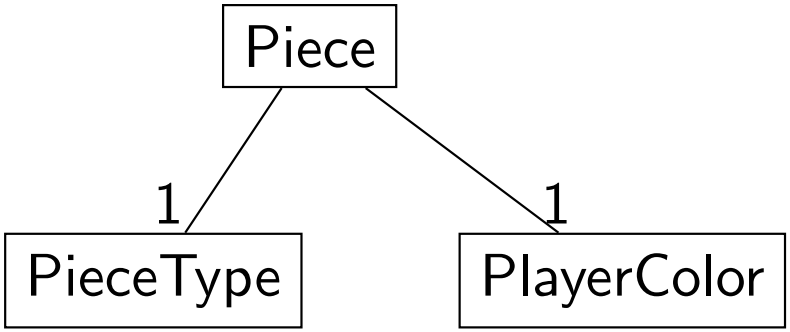
Spielfiguren



Die Klassen für Dreieck, Quadrat und Kreis werden nicht wirklich benötigt. Stattdessen können wir einen Aufzählungstyp benutzen. Auch für Größe und Farbe bieten sich Aufzählungstypen an:

```
enum Shape { TRIANGLE, SQUARE, CIRCLE}
enum Size  { LARGE, SMALL }
enum Color { WHITE, BLACK }
```

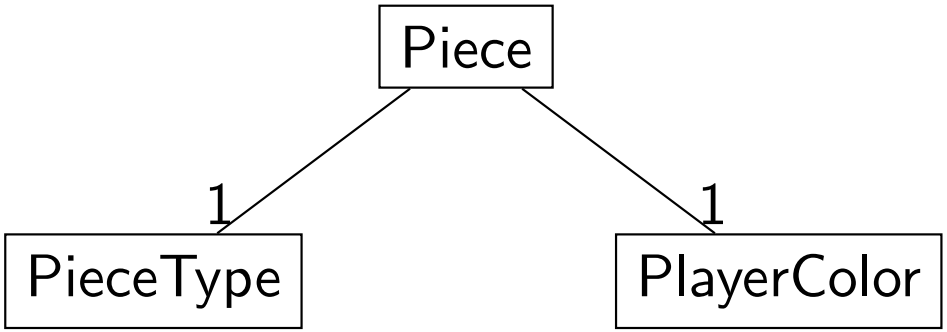
Spielfiguren – Implementierung



In unserer Referenzimplementierung haben wir Shape und Size zusammengefasst.

```
public enum PieceType {
    SMALLTRIANGLE, BIGTRIANGLE,
    SMALLSQUARE, BIGSQUARE,
    SMALLCIRCLE, BIGCIRCLE
}
public enum PlayerColor { WHITE, BLACK }
```

Spielfiguren – Implementierung



Die Klasse Piece enthält die beiden Aufzählungstypen als Komponenten.

```
public class Piece {
    PlayerColor player;
    PieceType pieceType;
    ...
}
```

Spielbrett

Ein Spielbrett muss sich die Figuren merken. Es gibt folgende Möglichkeiten.

- Spielbrett merkt sich eine Liste von Figuren, die Figuren merken sich ihre Position.

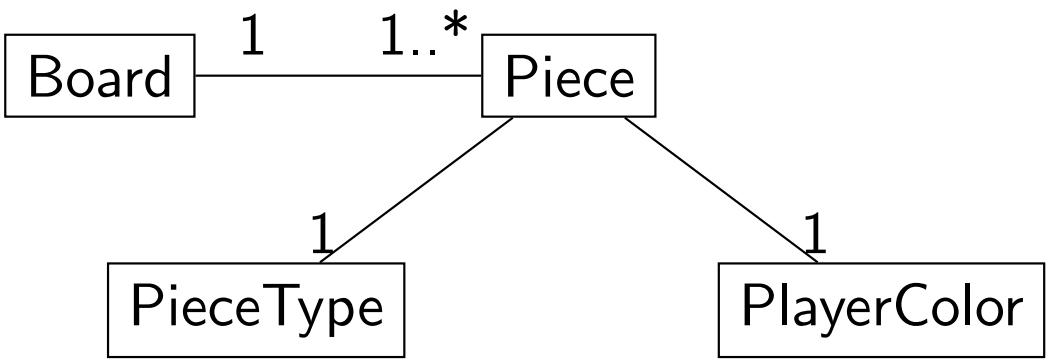
```
class Board { ArrayList<Piece> pieces; ...}
class Piece { int col, row;
    PieceType type; PlayerColor color; ...}
```

- Spielbrett merkt sich die Figuren in einem Array, wobei die Indices die Position angeben.

```
class Board { Piece[][] pieces; ...}
class Piece {
    PieceType type; PlayerColor color; ...}
```

Wir haben uns für die zweite Möglichkeit entschieden.

Spielbrett – Implementierung




```
class Board {Piece[][] pieces; ...}
class Piece {PieceType type; PlayerColor color;}
```

Was soll im Feld `pieces[r][c]` stehen, wenn an der Stelle `(r,c)` keine Figur steht?
Wieder zwei Möglichkeiten.

- Man benutzt die `null`-Referenz.
- Man baut eine Pseudofigur, die ein leeres Feld darstellt.

```
class NoPiece extends Piece {...}
```

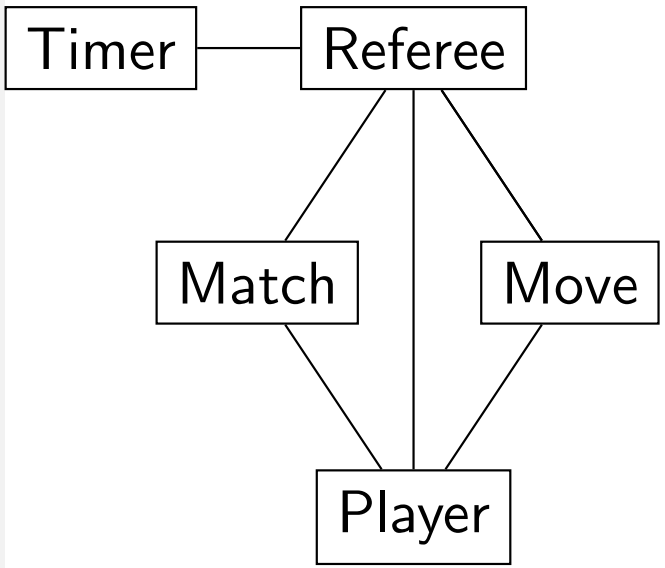
Wir haben uns für die zweite Möglichkeit entschieden.

Spieler-Interface

Es sollen zwei Spieler in einer Partie gegeneinander antreten.
Wir benutzen ein Interface, um verschiedene Implementierungen eines Spielers zu erlauben. Das Interface muss es ermöglichen, dem Spieler seine Farbe mitzuteilen, die Züge abzufragen und die Züge des Gegners mitzuteilen. Außerdem möchten wir dem Spieler ein Schiedsrichter-Objekt übergeben, über das der Spieler an weitere Funktionalität kommt.

```
public interface Player {
    public void init(
        PlayerColor color,
        Referee referee);

    public Move getMove(
        Move opponentMove);
}
```



Designentscheidungen beim Spieler-Interface

Weil das Interface `Player` von allen Spielprogrammen benutzt werden muss, haben wir versucht die Abhängigkeiten klein zu halten.

- Das `Board` wird nicht von der Schnittstelle referenziert. Jeder darf seine eigene Implementierung für das Spielbrett benutzen.
- Stattdessen ist nur `Move` Teil der Schnittstelle.
- Die Spielerfarbe wird in der `init`-Methode mitgeteilt.
- Die gegnerischen Züge werden bei `getMove` übergeben.
- Es gibt ein Schiedsrichter-Interface über das der Spieler mehr Informationen (z.B. Rechenzeit) bekommen kann.

Die Zug-Klasse

Die Klasse `Move` gehört auch zur Schnittstelle. Deshalb haben wir sie einfach gehalten:

- Der Konstruktor `Move(int oldLine, int oldColumn, int newLine, int newColumn, boolean remis)` erzeugt einen neuen Zug.
  Wenn man Remis anbietet muss man trotzdem einen gültigen Zug mitgeben.
- `getOldLine()`..`getNewColumn()` liefern den entsprechenden Wert.
- `isOfferingRemis()` liefert true, wenn der Zug Remis anbietet.

Objekte der Klasse Move sind unveränderlich (immutable). Man kann die Werte nur im Konstruktor setzen, danach bleiben sie fest. Man kann auch von der Klasse `Move` nicht erben.

Schiedsrichter-Interface

`public interface Referee` stellt weitere Methoden für die Spieler zur Verfügung

- `getRemainingTime()` liefert die Rechenzeit, die das Programm (für alle Züge) noch zur Verfügung hat, zurück. Wir werden wohl zehn Minuten Rechenzeit erlauben.
- `getOpponentRemainingTime()` liefert die Rechenzeit, die das Programm (für alle Züge) zur Verfügung hat zurück.
- `isValid(Move move)` kann benutzt werden, um einen Zug auf Gültigkeit zu testen.
- `writeMessage(String msg)` kann für Debuggingzwecke benutzt werden.