

# Dokumentation für das Damespiel-Projekt

Ange Dongmo Keunang , MatrikelNummer : 5525915, Technische Hochschule Mittelhessen

## Übersicht

Das Damespiel ist eine Java-Anwendung, die ein klassisches Damebrettspiel implementiert. Das Projekt umfasst die Spielmechanik, KI-Steuerung (Minimax-Algorithmus mit Alpha-Beta-Pruning), Browser-Integration zur Visualisierung und Unterstützung für Mehrfachzüge (inklusive Damenregeln). Die Anwendung ermöglicht es Spielern, gegen eine KI anzutreten oder über eine JShell-Schnittstelle zu interagieren.

---

## Klassen und Methoden

### Klasse Move

Verantwortlich für die Validierung und Berechnung möglicher Züge und Sprünge.

Methode	Beschreibung
<code>isValidPosition(int x, int y)</code>	Überprüft, ob die gegebene Position (x, y) innerhalb der Grenzen des Spielbretts liegt. Gibt <code>true</code> zurück, wenn die Position gültig ist, andernfalls <code>false</code> .

Clerk In Java Prototype

Methode	Beschreibung
<code>isValidMove(int x, int y)</code>	Prüft, ob ein Zug auf die Position (x, y) möglich ist. Ein Zug ist gültig, wenn die Position innerhalb des Bretts liegt und das Zielfeld leer ist. Gibt <code>true</code> zurück, wenn der Zug gültig ist, andernfalls <code>false</code> .

	Gibt eine Liste aller möglichen Züge für die Figur an der Position (fromX, fromY) zurück. Unterscheidet zwischen normalen Steinen und Damen. Normale Steine können sich nur vorwärts bewegen, während Damen sich in alle Richtungen bewegen können.
getPossibleMoves(int fromX, int fromY)	
getAllCaptures(int currentPlayer)	Sammelt alle möglichen Sprünge (Captures) für den aktuellen Spieler (currentPlayer). Gibt eine Map zurück, die für jede Figur des Spielers eine Liste aller möglichen Sprungpfade enthält.
exploreCaptures(int fromX, int fromY, int piece, List<List<Integer>> captures, List<Integer> positions)	Rekursive Methode zur Erkundung von Mehrfachsprüngen. Sie untersucht alle möglichen Sprungpfade für eine Figur an der Position (fromX, fromY) und fügt gültige Pfade zur Liste captures hinzu. Gibt true zurück, wenn weitere Sprünge möglich sind, andernfalls false.
handleSimplePieceCaptures(int fromX, int fromY, int piece, List<Integer> positions, List<List<Integer>> captures)	Behandelt Sprünge für normale Steine. Überprüft alle möglichen Sprungrichtungen und fügt gültige Sprünge zur Liste captures hinzu. Gibt true zurück, wenn weitere Sprünge möglich sind, andernfalls false.
Methode	Beschreibung
localhost:50001 27.01.25, 20:30	handleQueenCaptures(int fromX, int fromY, int piece, List<Integer> positions, List<List<Integer>> captures)

3/25

## Klasse Checkers

Verwaltet den Spielzustand, die Browser-Integration und Spielerinteraktion.

Methode	Beschreibung
resetBoard()	Initialisiert das Spielbrett mit der Startaufstellung. Normale Steine werden auf den ersten drei Reihen für Spieler 1 und auf den letzten drei Reihen für Spieler 2 platziert. Die Sets Black_Pieces und Other_Pieces werden entsprechend aktualisiert.
updateBrowser(int fromX, int fromY, int toX, int toY, int currentPlayer)	Aktualisiert die Browseransicht mit dem aktuellen Spielstand. Die Methode sendet den aktuellen Zustand des Bretts an das Frontend, um die grafische Darstellung zu aktualisieren.

<code>moves(int fromX, int fromY, int toX, int toY)</code>	Führt einen Zug aus und prüft auf Gültigkeit und Sprungpflicht. Die Methode überprüft, ob der Zug gültig ist und ob ein Sprung (Capture) möglich ist. Falls ein Sprung möglich ist, wird dieser ausgeführt. Andernfalls wird ein normaler Zug ausgeführt.
--	---

## Clerk in Java Prototype

Methode	Beschreibung
<code>executeMove(int fromX, int fromY, int toX, int toY)</code>	Bewegt eine Figur von (fromX, fromY) nach (toX, toY) und aktualisiert die Sets <code>Black_Pieces</code> und <code>Other_Pieces</code> . Falls eine Figur die gegnerische Grundlinie erreicht, wird sie zur Dame befördert.
<code>undoMove(int fromX, int fromY, int toX, int toY)</code>	Macht einen Zug rückgängig. Die Methode wird für den Minimax-Algorithmus benötigt, um Züge zu simulieren und anschließend den ursprünglichen Zustand wiederherzustellen.
<code>handleCapture(int fromX, int fromY, int toX, int toY)</code>	Behandelt Sprünge (Captures) für eine Figur. Die Methode überprüft, ob ein Sprung möglich ist, und führt diesen aus. Falls ein Sprung ausgeführt wird, wird der Spieler gewechselt.
<code>executeCapture(List&lt;Integer&gt; capture, int fromX, int fromY, int toX, int toY)</code>	Führt einen Sprung (Capture) aus. Die Methode entfernt die geschlagenen Figuren vom Brett und aktualisiert die Sets <code>Black_Pieces</code> und <code>Other_Pieces</code> .
<code>undoCapture(int fromX, int fromY, int toX, int toY)</code>	Macht einen Sprung (Capture) rückgängig. Die Methode stellt die geschlagenen Figuren wieder auf dem Brett her und aktualisiert die Sets <code>Black_Pieces</code> und <code>Other_Pieces</code> .
<code>isGameOver()</code>	Überprüft, ob das Spiel beendet ist. Gibt <code>true</code> zurück, wenn keine Figuren mehr für einen der Spieler auf dem Brett sind, andernfalls <code>false</code> .
Methode	Beschreibung
<code>destination(List&lt;Integer&gt; capture, int to)</code>	Überprüft, ob das Ziel eines Sprungpfads (Capture) mit der gegebenen Position <code>to</code> übereinstimmt. Gibt <code>true</code> zurück, wenn das Ziel gültig ist, andernfalls <code>false</code> .
<code>sendPossibleMoves(List&lt;int[]&gt; moves, String action)</code>	Sendet eine Liste möglicher Züge an das Frontend, um sie grafisch darzustellen.
<code>sendPossibleCaptures(Map&lt;Integer, List&lt;List&lt;Integer&gt;&gt;&gt; captures, String action, int number)</code>	Sendet eine Liste möglicher Sprünge (Captures) an das Frontend, um sie grafisch darzustellen.
<code>getPosition(int row, int col)</code>	Konvertiert eine Zeilen- und Spaltenposition in einen eindimensionalen Index.

<code>getCoordinates(int position)</code>	Konvertiert einen eindimensionalen Index zurück in eine Zeilen- und Spaltenposition.
<code>toString()</code>	Gibt eine textuelle Darstellung des aktuellen Spielbretts zurück. Die Methode wird hauptsächlich für Debugging-Zwecke verwendet.

**Weitere Hinweise:**

localhost:50001

27.01.25, 20:30

5/25

- **Browser-Integration:** Die Klasse `Checkers` ist eng mit dem Frontend verbunden. Sie verwendet die `Clerk`-Klasse, um mit dem Browser zu kommunizieren und das Spielbrett grafisch darzustellen.
- **Spielerwechsel:** Nach jedem Zug wird der aktive Spieler gewechselt. Falls der nächste Spieler die KI ist, wird die Methode `ia.playAIMove()` aufgerufen, um den Zug der KI auszuführen.
- **Rückgängig machen von Zügen:** Die Methoden `undoMove` und `undoCapture` werden verwendet, um Züge rückgängig zu machen. Dies ist insbesondere für den Minimax-Algorithmus wichtig, der verschiedene Züge simuliert und den ursprünglichen

Zustand wiederherstellen muss.

## Interaktion des Spiels: Klasse Checkers

Die Klasse `Checkers` ist für die Interaktion zwischen dem Spieler und dem Spielbrett verantwortlich. Sie verwaltet die Logik des Spiels, die grafische Darstellung des Bretts und die Kommunikation mit dem Backend. Hier ist eine detaillierte Erklärung der Interaktion:

---

### 1. Initialisierung des Spiels

- **Konstruktor (constructor)**

- Der Konstruktor initialisiert das Spielbrett, lädt die Bilder der Spielsteine und setzt die Event-Listener für die Mausklicks auf das Canvas.
  - Das Spielbrett wird als 8x8-Grid dargestellt, und jede Zelle hat eine Größe von `cellSize`.
  - Die Methode `initBoard()` wird aufgerufen, um das Brett mit den Startpositionen der Spielsteine zu initialisieren.
- 

### 2. Spielbrett und Grafiken

- **initBoard()**

- Initialisiert das Brett mit den Startpositionen der Spielsteine. Die Spielsteine des Spielers (1) werden auf den unteren drei Reihen platziert, während die Spielsteine des Gegners (-1) auf den oberen drei Reihen platziert werden.

- **drawPiece(piece, x, y)**

- Zeichnet eine Spielstein auf dem Brett an der Position `(x, y)`. Die Methode verwendet die geladenen Bilder für die Spielsteine und Damen.

- `drawBoard(calling, board, from, to, currentPlayer)`
    - Zeichnet das gesamte Brett und die Spielsteine. Wenn `calling` auf `true` gesetzt ist, wird eine Nachricht an das Backend gesendet, um den aktuellen Spielzustand zu aktualisieren.
    - Die Methode hebt auch die Zellen hervor, von denen und zu denen ein Zug ausgeführt wurde.
- 

### 3. Spielerinteraktion

- **Event-Listener für Mausklicks**
    - Der Event-Listener erfasst Klicks auf das Canvas und berechnet die Position der angeklickten Zelle.
    - Je nach Zustand des Spiels wird entweder `firstClick()` oder `secondClick()` aufgerufen.
  - `firstClick(from, to, index)`
    - Wird aufgerufen, wenn der Spieler auf eine Zelle klickt, die einen Spielstein enthält.
    - Überprüft, ob der Spielstein dem aktuellen Spieler gehört (`Turn(index)`).
    - Wenn ein Sprung (Capture) möglich ist, werden die möglichen Sprungpfade hervorgehoben.
    - Wenn kein Sprung möglich ist, werden die möglichen normalen Züge angezeigt.
  - `secondClick(from, to, index)`
    - Wird aufgerufen, wenn der Spieler auf eine leere Zelle klickt.
    - Überprüft, ob der Zug gültig ist (entweder ein normaler Zug oder ein Sprung).
    - Wenn der Zug gültig ist, wird eine Nachricht an das Backend gesendet, um den Zug auszuführen.
- 

### 4. Kommunikation mit dem Backend

- `call(message)`

- Sendet eine Nachricht an das Backend, um den aktuellen Spielzustand zu aktualisieren oder einen Zug auszuführen.
  - Die Nachricht enthält Informationen wie die Positionen der Spielsteine oder den aktuellen Spieler.
  - `waitForMove()`
    - Wartet auf eine Antwort vom Backend, nachdem ein Zug gesendet wurde.
    - Die Methode gibt ein `Promise` zurück, das aufgelöst wird, sobald das Backend antwortet.
  - `manageMove(array)`
    - Verarbeitet die Antwort vom Backend, die die möglichen Züge enthält.
    - Die möglichen Züge werden auf dem Brett hervorgehoben.
  - `manageCapture(map, turn)`
    - Verarbeitet die Antwort vom Backend, die die möglichen Sprünge (Captures) enthält.
    - Die Methode aktualisiert die interne Map mit den möglichen Sprungpfaden und hebt die entsprechenden Zellen hervor.
- 

### 5. Sprungpflicht und Zugpflicht

- `mandatoryCapture(map)`
    - Überprüft, ob ein Sprung (Capture) für den aktuellen Spieler verpflichtend ist.
    - Wenn ein Sprung möglich ist, werden die entsprechenden Zellen hervorgehoben.
  - `handleNormalMove(from, to, index)`
    - Verarbeitet normale Züge (ohne Sprünge).
    - Sendet eine Nachricht an das Backend, um den Zug zu validieren und die möglichen Züge anzuzeigen.
-

## 6. Grafische Hervorhebungen

- `highlightCell(index, color)`
    - Hebt eine Zelle auf dem Brett mit einer bestimmten Farbe hervor.
    - Wird verwendet, um mögliche Züge oder Sprünge anzuzeigen.
  - `lightCell(index, color, preserveImage)`
    - Zeichnet eine farbige Überlagerung auf einer Zelle, während das Bild des Spielsteins (falls vorhanden) beibehalten wird.
- 

## 7. Zusammenfassung der Interaktion

1. Der Spieler klickt auf einen Spielstein (`firstClick()`).
  2. Das Spiel zeigt die möglichen Züge oder Sprünge an.
  3. Der Spieler klickt auf eine leere Zelle (`secondClick()`).
  4. Das Spiel überprüft, ob der Zug gültig ist, und sendet eine Nachricht an das Backend.
  5. Das Backend verarbeitet den Zug und sendet eine Antwort zurück.
  6. Das Spiel aktualisiert das Brett und den Spielzustand basierend auf der Antwort.
- 

### Beispielablauf:

1. **Spielstart:** Das Brett wird initialisiert, und die Spielsteine werden platziert.
2. **Spieler 1 klickt auf einen Spielstein:** Die möglichen Züge oder Sprünge werden angezeigt.
3. **Spieler 1 klickt auf eine leere Zelle:** Der Zug wird validiert und an das Backend gesendet.
4. **Backend antwortet:** Das Spiel aktualisiert das Brett und wechselt den Spieler.
5. **Spieler 2 (KI) führt einen Zug aus:** Das Spiel aktualisiert das Brett erneut.

Diese Interaktion wiederholt sich, bis das Spiel beendet ist.

localhost:50001

9/25

27.01.25, 20:30

Clerk in Java Prototype

## Erklärung der Klasse IA (Künstliche Intelligenz für Dame-Spiel)

Die Klasse `IA` implementiert eine künstliche Intelligenz (KI) für das Dame-Spiel. Sie verwendet den Minimax-Algorithmus mit Alpha-Beta-Pruning, um den besten Zug für die KI zu berechnen. Im Folgenden wird die Funktionsweise detailliert beschrieben.

### 1. Initialisierung und Konstruktor

`IA(Checkers game)`

- Initialisiert die KI mit einer Referenz auf das `Checkers`-Spiel und das `Move`-Objekt.
  - Die KI verwendet diese Referenzen, um auf das Spielbrett und die Spielregeln zuzugreifen.
- 

### 2. Bewertung des Spielbretts

`evaluateBoard(int[] board)`

- Bewertet das aktuelle Spielbrett basierend auf der Position und dem Typ der Spielsteine.

**Bewertungskriterien:**

#### 1. Spielsteine:

- Normaler Spielstein (Pion): Wert von 3.
- Dame: Wert von 5.

#### 2. Bedrohungen:

- Bedrohter Spielstein (kann im nächsten Zug geschlagen werden): Strafwert von -3

localhost:50001

10/25

```
isThreatened(int position)
```

- Überprüft, ob ein Spielstein an der gegebenen Position bedroht ist.
  - Ein Spielstein ist bedroht, wenn ein gegnerischer Spielstein ihn im nächsten Zug schlagen kann.
- 

### 3. Minimax-Algorithmus mit Alpha-Beta-Pruning

```
minimax(int depth, boolean isMaximizingPlayer, int alpha, int beta)
```

- Berechnet den besten Zug für die KI.

**Parameter:**

- **depth**: Aktuelle Tiefe der Rekursion.
- **isMaximizingPlayer**: Gibt an, ob der Spieler maximiert (KI) oder minimiert (Gegner).
- **alpha und beta**: Begrenzungen für das Alpha-Beta-Pruning.

**Funktionsweise:**

1. Bei maximaler Tiefe oder Spielende: Bewertet das Brett.
  2. Generiert alle möglichen Züge für den aktuellen Spieler.
  3. Simuliert jeden Zug und ruft den Algorithmus rekursiv auf.
  4. Wählt den besten Zug basierend auf der Bewertung aus.
  5. Verwendet Alpha-Beta-Pruning zur Effizienzsteigerung.
- 

### 4. Dynamische Anpassung der Suchtiefe

11/25

```
calculateDynamicDepth()
```

localhost:50001

27.01.25, 20:30

- Passt die Suchtiefe basierend auf der Anzahl der verbleibenden Spielsteine der KI an.

**Regeln:**

1. Wenige Spielsteine (Endspiel): **Höhere Suchtiefe (4)**.
  2. Mittlere Anzahl von Spielsteinen (Mittelspiel): **Mittlere Suchtiefe (3)**.
  3. Viele Spielsteine (Anfangsspiel): **Niedrigere Suchtiefe (2)**.
- 

### 5. Generierung möglicher Züge

```
getNormalMoves(int currentPlayer)
```

- Gibt eine Liste aller möglichen normalen Züge für den aktuellen Spieler zurück.

```
getCaptureMove(int currentPlayer)
```

- Gibt eine Liste aller möglichen Sprünge (Captures) für den aktuellen Spieler zurück.

```
generateToPositionAfterCaptures(List<Integer> capture)
```

- Generiert mögliche Endpositionen nach einem Sprung.
    - **Damen**: Alle möglichen Endpositionen entlang des Sprungpfads.
    - **Normale Spielsteine**: Nur die letzte Position des Sprungpfads.
- 

localhost:50001

### 6. Bestimmung des besten Zugs

12/25

- Sucht den besten Zug basierend auf der Minimax-Bewertung.
  - Gibt den Zug mit der höchsten Bewertung zurück.
- 

## 7. Ausführung des KI-Zugs

`playAIMove()`

- Führt den besten Zug der KI aus.
1. Ruft `bestMove` auf, um den besten Zug zu bestimmen.
  2. Führt den Zug aus und aktualisiert das Spielbrett.
  3. Schaltet den Spieler um und aktualisiert den Spielstatus.
- 

## 8. Zusammenfassung der Funktionsweise

1. **Initialisierung:** Die KI wird mit dem Spielbrett und den Regeln initialisiert.
  2. **Bewertung:** Das Brett wird basierend auf der Position der Spielsteine bewertet.
  3. **Minimax:** Der Algorithmus berechnet den besten Zug.
  4. **Dynamische Tiefe:** Die Suchtiefe wird dynamisch angepasst.
  5. **Zuggenerierung:** Alle möglichen Züge werden analysiert.
  6. **Bester Zug:** Der beste Zug wird gewählt.
  7. **Zugausführung:** Der Zug wird gespielt, und das Spielbrett wird aktualisiert.
- 

### Beispielablauf:

13/25

1. **Spielstart:** KI wird initialisiert und bereitet den ersten Zug vor.

localhost:50001

27.01.25, 20:30

2. **Zugberechnung:** KI generiert alle möglichen Züge und bewertet sie mit Minimax.
3. **Zugausführung:** KI führt den besten Zug aus.
4. **Spielerwechsel:** Der Gegner ist an der Reihe, und der Prozess wiederholt sich.

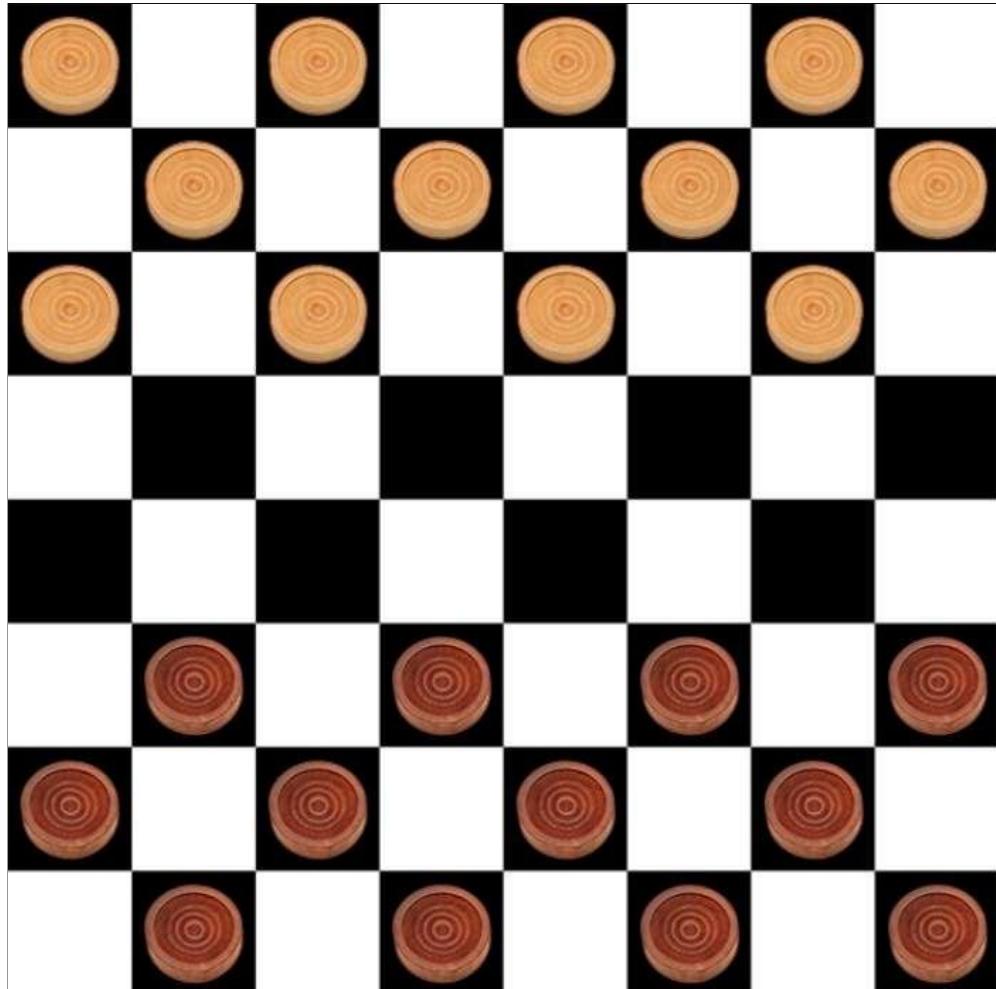
Dieser Zyklus wiederholt sich, bis das Spiel beendet ist. Die KI versucht, durch optimale Züge das Spiel zu gewinnen.

---

### Szenario 1: Erstellung des Spielbretts

```
Checkers game = new Checkers();
```

Das Ergebnis sieht dann so aus:



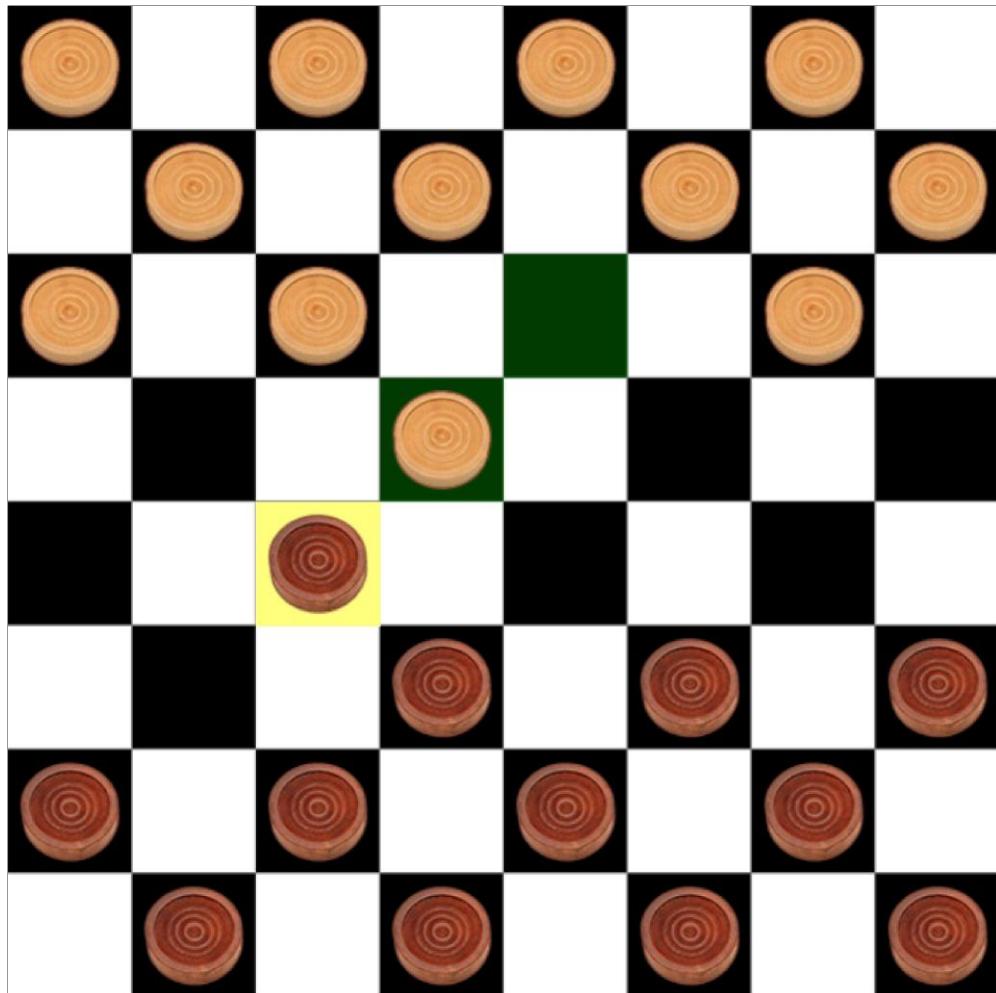
15/25

localhost:50001

## Szenario 2: Bewegung von Steinen

```
Checkers game = new Checkers();  
game.moves(5, 1, 4, 2);
```

Der Spieler bewegt das Stein von (5,1) nach (4,2). Das Ergebniss sieht so aus :



17/25

localhost:50001

27.01.25, 20:30

Clerk in Java Prototype

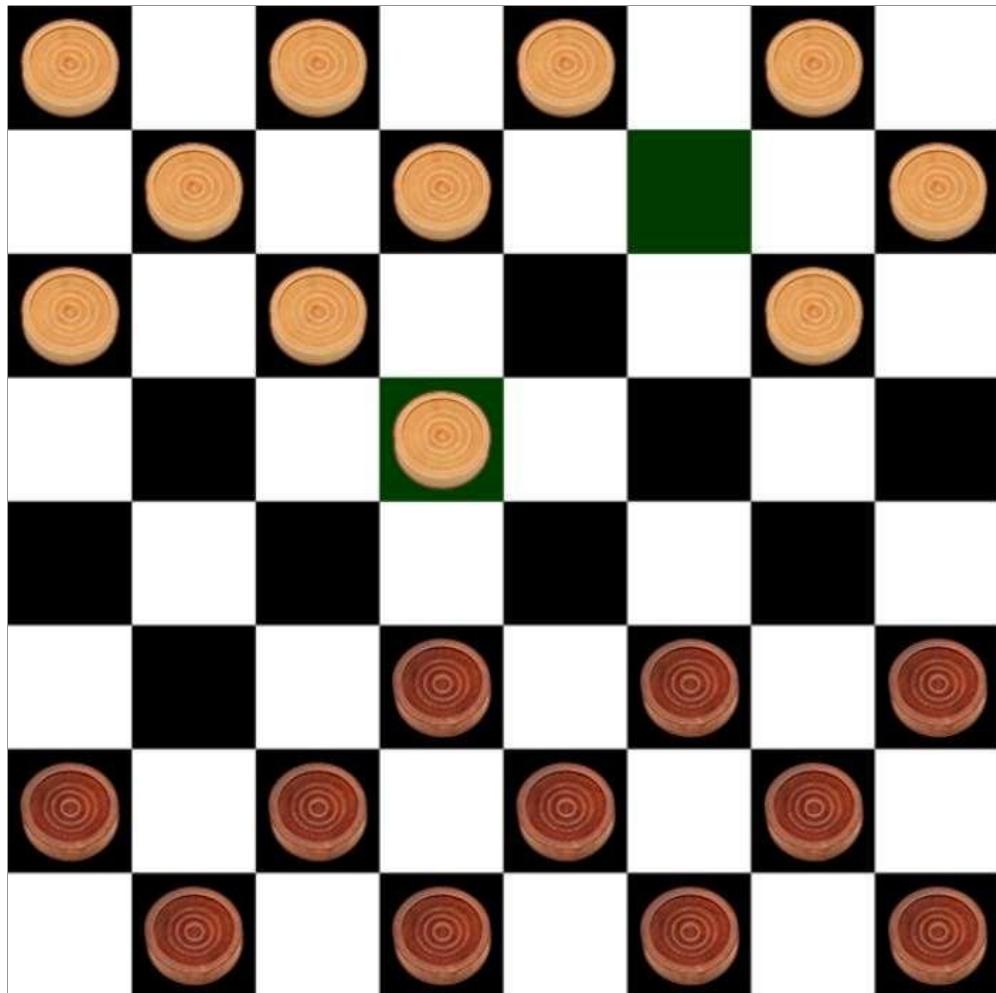
### Szenario 3: schlagen von Steinen

```
Checkers games = new Checkers();
games.moves(5, 1, 4, 2); // Spieler bewegt von (5,1) nach (4,2)
//Mit der Interaktivitt sieht das Schlagen besser aus.
```

In diesem Beispiel spiele ich gegen die KI, die auf meine Zge reagiert. Es entsteht eine Situation, in der ich einen Stein der KI schlage, und die KI im Gegenzug einen meiner Steine schagt. Sie knnen beobachten, wie die geschlagenen Steine vom Spielbrett entfernt werden.

localhost:50001

18/25



19/25

localhost:50001

27.01.25, 20:30

Clerk in Java Prototype

#### Szenario 4: Umwandlung in eine Dame

---

##### Erklärung des Szenarios:

###### 1. Umwandlung in eine Dame:

- Ein Spielstein erreicht die gegnerische Grundlinie (in diesem Fall Zeile 7).
- Der Stein wird automatisch in eine Dame umgewandelt.

###### 2. Erweiterte Bewegungsmöglichkeiten:

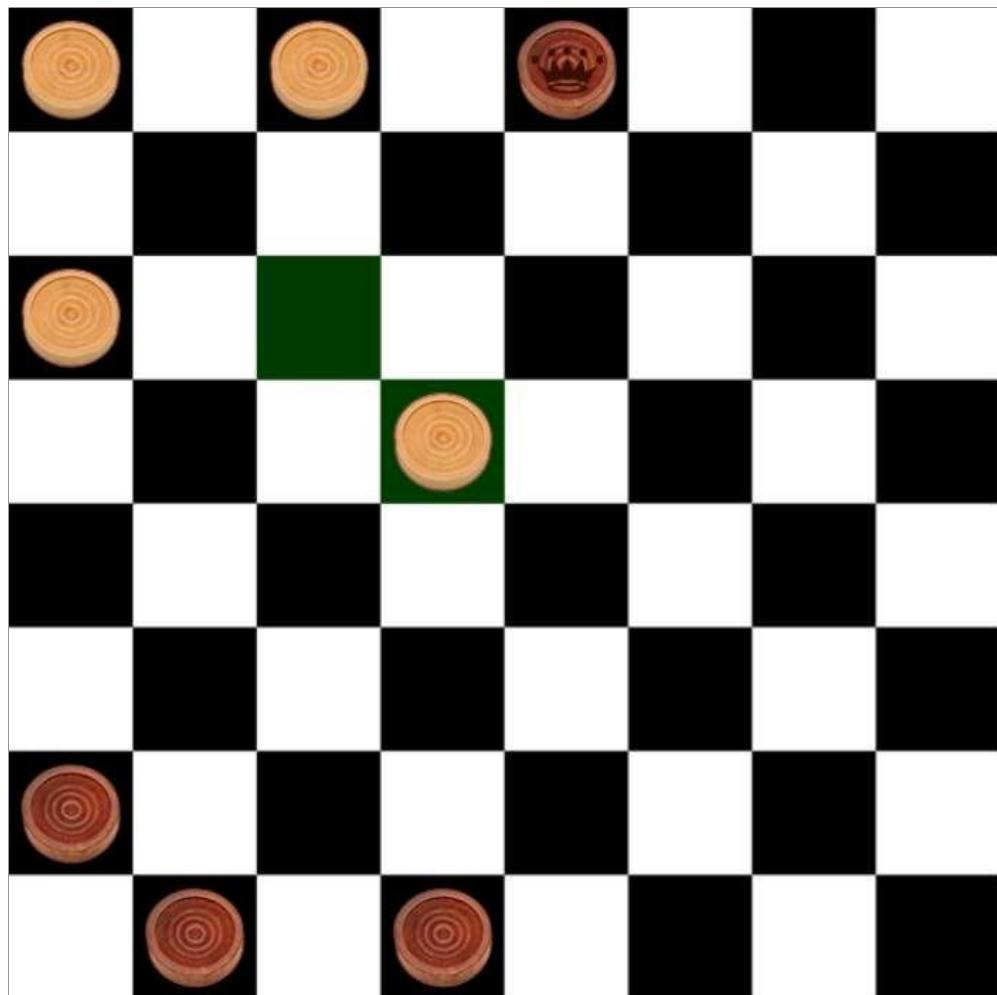
- Eine Dame kann sich in alle diagonalen Richtungen bewegen, sowohl vorwärts als auch rückwärts.
- Dies gibt der Dame mehr strategische Möglichkeiten im Spiel.

###### 3. Aktualisierung des Spielfelds:

- Das Spielfeld wird grafisch aktualisiert, um die Umwandlung des Steins in eine Dame anzuzeigen.
  - Die Dame wird durch ein spezielles Symbol oder eine andere Farbe dargestellt.
- 

Wir spielen das Spiel Schritt für Schritt und lassen es in eine zufällige Situation geraten, um die Umwandlung eines Steins in eine Dame zu demonstrieren.

```
Checkers Game = new Checkers();
```



21/25

## Szenario 5: Spielen gegen die KI

### Erklärung des Szenarios:

#### 1. KI berechnet und führt Züge aus:

- Die KI verwendet den Minimax-Algorithmus mit Alpha-Beta-Pruning, um den besten Zug zu berechnen.
- Der Zug der KI wird automatisch auf dem Spielfeld ausgeführt.

#### 2. Abwechselndes Spielen:

- Der menschliche Spieler und die KI wechseln sich ab, wobei jeder einen Zug macht.
- Nach jedem Zug wird das Spielfeld aktualisiert, um den neuen Spielzustand anzuzeigen.

#### 3. Sieg-Bedingungen:

- Das Spiel überprüft kontinuierlich, ob ein Spieler gewonnen hat.
  - Ein Spieler gewinnt, wenn der Gegner keine Steine mehr hat oder keine gültigen Züge mehr möglich sind.
  - Der Gewinner wird angezeigt, sobald das Spiel beendet ist.
- 

## Methoden zur Anzeige des Spielergebnisses

### 1. `message()` (Java)

- **Rolle:** Überprüft, ob das Spiel beendet ist, und zeigt eine Sieges- oder Unentschieden-Nachricht an.

- **Funktionen:**

- Bestimmt den Gewinner oder ob das Spiel unentschieden ist.
- Sendet die Nachricht über `Clerk.call` an die JavaScript-Funktion `drawMessage`.
- Setzt das Spiel bei Bedarf zurück.

### 2. `drawMessage()` (JavaScript)

- **Rolle:** Zeigt die Nachricht auf dem Canvas und in einem Alert-Fenster an.

- **Funktionen:**

- Zeichnet den Text in der Mitte des Canvas.
- Verwendet eine rote Schriftart und eine große Schriftgröße für bessere Sichtbarkeit.
- Zeigt zusätzlich ein Alert-Fenster an, um sicherzustellen, dass der Spieler die Nachricht sieht.

## Zusammenarbeit

- **Java → JavaScript:** Die Methode `message()` sendet die Nachricht an `drawMessage()` zur Anzeige.
  - **Frontend:** `drawMessage()` übernimmt die visuelle Anzeige auf dem Canvas und im Alert-Fenster.
- 

## Methoden und Erklärungen in Tabellenform

Methode	Bedeutung
message()	Überprüft das Spielende und übermittelt die Nachricht an drawMessage für die Anzeige.
drawMessage()	Zeigt die Nachricht visuell auf dem Canvas und in einem Alert-Fenster an.

Am Ende sieht das Ergebniss so aus :

