Funktionale und objektorientierte Programmierkonzepte
Übungsblatt 01

Prof. Karsten Weihe

Wintersemester 22/23	v1.1
Themen:	Programmieren in Java mit Hilfe von FopBot
Relevante Foliensätze:	01a und 01b
Abgabe der Hausübung:	04.11.2022 bis 23:50 Uhr

Hausübung 01 — Gesamt: 24 Punkte

Little Checkers

> **Beachten Sie die Seite *Verbindliche Anforderungen für alle Abgaben* im Moodle-Kurs.**
>
> Verstöße gegen verbindliche Anforderungen führen zu Punktabzügen und können die korrekte Bewertung Ihrer Abgabe beeinflussen. Sofern vorhanden, müssen die in der Vorlage mit `TODO` markierten `crash`-Aufrufe entfernt werden. Andernfalls wird die jeweilige Aufgabe nicht bewertet.

Die für diese Hausübung relevanten Verzeichnisse sind `src/main/java/h01` und ggf. `src/test/java/h01`.

Sofern Sie das entsprechende Vorwissen haben, dürfen Sie Arrays verwenden und weitere Methoden erstellen, um Ihren Code besser zu strukturieren, indem Sie Ihre Methoden aus diesen Bereichen heraus aufrufen.

Auf diesem und weiteren Übungsblättern finden Sie häufig Formulierungen, dass Entitäten wie `robot` dieses oder jenes *tun*. Solche vermenschlichenden Formulierungen sind natürlich nicht wirklich korrekt, dafür aber einfacher und intuitiver und daher allgemein sehr beliebt.

Einleitung

Mit diesem Übungsblatt implementieren Sie ein kleines Spiel, dessen Idee auf dem Brettspiel *Dame* (engl. *Checkers*) basiert. Die tatsächliche Umsetzung unterscheidet sich jedoch stark.

Auf dem *Spielbrett* befinden sich fünf schwarze Roboter des Teams *Schwarz* und ein weißer Roboter des Teams *Weiß*. Das Spielbrett wird mittels der Ihnen bereits bekannten Welt der Roboter dargestellt.

Kern Ihrer Lösung ist die Implementation einzelner Bestandteile einer `while`-Schleife, welche im Folgenden auch *Hauptschleife* genannt wird.

In jedem Durchlauf der Hauptschleife macht einer der nicht geschlagenen schwarzen Steine einen Zug. Nach jedem Zug eines nicht geschlagenen schwarzen Steins wird überprüft, ob der weiße Stein einen der schwarzen Steine schlagen kann. Die Hauptschleife und damit das ganze Spiel ist zu Ende, sobald der weiße Stein alle schwarzen Steine geschlagen hat oder alle schwarzen Steine ihre Münze abgelegt haben.

Darstellung von Steinen

Die Steine werden mittels der Ihnen bereits bekannten Roboter dargestellt.

Damit die Roboter beider Teams unterschieden werden können, sollen die Roboter unterschiedlichen Roboter-Familien angehören. Roboter verschiedener Roboter-Familien unterscheiden sich in ihrem Aussehen: Die Roboter von Team Schwarz gehören zur Familie der *Black Square Robots*, der Roboter von Team Weiß gehört zur Familie der *White Square Robots*. Wir bezeichnen die Roboter beider Familien im Folgenden auch als *schwarze Steine* bzw. *weiße Steine*.

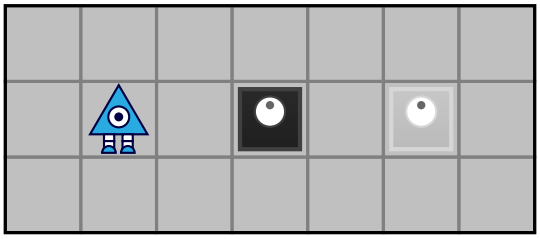

Abbildung 1: Auswahl von Roboter-Familien

Um einen Roboter einer anderen Familie als der Ihnen bereits bekannten Familie `TRIANGLE_BLUE` zu konstruieren, nutzen Sie einen beliebigen Konstruktor von `Robot` mit Parameter vom Typ `RobotFamily`. Bei `RobotFamily` handelt es sich wie bei `Direction` um eine *Enumeration* – mit dem Unterschied, dass `RobotFamily` statt der Blickrichtungen die Roboter-Familien aufzählt, darunter (Abbildung 1, von links nach rechts) `TRIANGLE_BLUE`, `SQUARE_BLACK` (Black Square Robots) und `SQUARE_WHITE` (White Square Robots).

Ein Stein wird genau dann als geschlagen bezeichnet, wenn dieser ausgeschaltet ist. Square Robots haben die Eigenschaft, dass sie im ausgeschalteten Zustand ihre Augen verschlossen halten.

Beispiel:

Einen Roboter der Familie `RobotFamily.SQUARE_WHITE` können Sie beispielsweise wie folgt konstruieren:

```
1  Robot robot = new Robot(3, 1, Direction.UP, 8, RobotFamily.SQUARE_WHITE);
```

Einstellungen

In Verzeichnis `src/main/resources` finden Sie die Datei `checkers.properties`: In dieser können Sie von uns festgelegten Schlüsseln Werte zuweisen, welche in der Vorlage zum Start Ihres Programmes eingelesen und in den gleichnamigen Klassenkonstanten gespeichert werden. Wenn auf diesem Übungsblatt die Rede davon ist, dass ein Wert *eingelesen* wird, ist gemeint, dass ein Wert aus der Datei `checkers.properties` eingelesen wird.

Testen Sie Ihr Programm systematisch durch, indem Sie die gegebenen Werte abändern.

Verbindliche Anforderung:

Ihr Programm muss neben den vordefinierten auch mit allen anderen *sinnvollen* Werten funktionieren!

Dieser Grad an Flexibilität wird für uns bei allen möglichen Eingabedaten, die prinzipiell von Programmlauf zu Programmlauf variieren könnten, aber in jedem einzelnen Programmlauf konstant sind, selbstverständlich sein: nicht den Quelltext ändern, nur die Eingabedaten. Diese Eingabedaten können dann wie hier aus einer Datei kommen oder am Computer abgefragt werden, aus einer Datenverbindung oder woher auch immer kommen.

H1: Initialisierung — 9 Punkte

Erinnerung:

Beachten Sie bei *jedem* Übungsblatt, dass nach Bearbeitung einer Aufgabe die jeweiligen mit `// TODO` markierten Aufrufe von `crash` entfernt werden müssen. Andernfalls funktioniert Ihre Implementation nicht korrekt und die jeweiligen Teile Ihres Quelltextes werden *nicht* bewertet!

Im Gegensatz zum tatsächlichen Damespiel kann unser Spielbrett jede beliebige Größe größer oder gleich 3×3 annehmen. Die Breite und Höhe des Spielbretts wird in die Konstanten `NUMBER_OF_ROWS` und `NUMBER_OF_COLUMNS` eingelesen und in der Vorlage der Welt bereits zugewiesen.

In Klasse `Checkers` sind für die fünf schwarzen Steine die Attribute `blackStone0`, ..., `blackStone4` und für den weißen Stein das Attribut `whiteStone` vom Typ `Robot` deklariert.

In den folgenden beiden Teilaufgaben implementieren Sie die Initialisierung dieser Steine.

H1.1: Initialisierung des weißen Steins — 4 Punkte

Zunächst implementieren Sie die Platzierung des weißen Steins in der Methode `initWhiteStone`.

Das Attribut `whiteStone` soll bei Aufruf von `initWhiteStone` mit einem Roboter der Familie `SQUARE_WHITE` initialisiert werden, welcher keine Münzen besitzt. Die Position und die Richtung des weißen Steins `whiteStone` soll dabei pseudozufällig[1] so gewählt werden, dass die beiden folgenden Bedingungen erfüllt sind:

1. Der Position liegt innerhalb der Welt.
2. Die Summe aus Spaltenindex und Zeilenindex ist eine ungerade Zahl.

Anmerkung:

Die Erzeugung einer beliebig langen Folge pseudozufälliger Zahlen vom Typ `int` ist mit Werkzeugen der Java-Standardbibliothek möglich. Zum Beispiel liefert `ThreadLocalRandom.current().nextInt(1, 7)` eine pseudozufällig gewählte Zahl aus $\{1, \ldots, 6\}$, simuliert also einen normalen Würfel mit sechs Seiten.

Hinweis:

Zur pseudozufälligen Festlegung der initialen Richtung erzeugen Sie eine Zufallszahl $0 \ldots 3$. Diese speichern Sie zweckmäßigerweise in einer `int`-Variablen `randomDirectionValue`. Dann richten Sie noch eine Variable `randomDirection` vom Typ `Direction` ein und setzen ihren Wert abhängig vom Wert von `randomDirectionValue`, nämlich auf `UP` im Fall 0, `RIGHT` im Fall 1, `DOWN` im Fall 2 und `LEFT` im Fall 3. Bei der Einrichtung des zu initialisierenden Roboter-Objektes mit `new` setzen Sie dann die Variable `randomDirection` anstelle einer festen Richtung ein.

[1] Siehe https://de.wikipedia.org/wiki/Pseudozufall

H1.2: Initialisierung der schwarzen Steine — 5 Punkte

Nun implementieren Sie die Platzierung der schwarzen Steine in Methode `initBlackStones`. Gehen Sie davon aus, dass beim Aufruf von `initBlackStones` das Attribut `whiteStone` bereits initialisiert ist.

In Methode `initBlackStones` soll jedes der Attribute `blackStone0`, ..., `blackStone4` mit einem Roboter der Familie `SQUARE_BLACK` initialisiert werden. Die Position und die Richtung jedes schwarzen Steins soll wie die des weißen Steins `whiteStone` in H1.1 gewählt werden. Jedoch darf ein schwarzer Stein *nicht* auf dem Feld platziert werden, auf dem sich der bereits initialisierte weiße Stein befindet. Ein schwarzer Stein darf hingegen auch dann auf einem Feld platziert werden, wenn sich dort ein anderer schwarzer Stein befindet.

In Klasse `Checkers` finden Sie die eingelesenen Konstanten `MIN_NUMBER_OF_COINS` und `MAX_NUMBER_OF_COINS`, wobei Sie davon ausgehen können, dass $1 \leq$ `MIN_NUMBER_OF_COINS` $\leq$ `MAX_NUMBER_OF_COINS`. Jeder der schwarzen Steine soll eine pseudozufällige Anzahl an Münzen besitzen, welche zwischen `MIN_NUMBER_OF_COINS` und `MAX_NUMBER_OF_COINS` (jeweils einschließlich) liegt.

Ausblick:

Ärgern Sie sich vielleicht, dass Sie `blackStone0`, ..., `blackStone4` jeweils einzeln auf genau dieselbe Art behandeln müssen und den Code zur Behandlung eines Steins nicht einfach nur einmal, nämlich in einer Schleife $0, \ldots, 4$ hinschreiben können? Ab dem nächsten Übungsblatt lernen Sie, wie Sie dies mit *Arrays* tun können.

H1.3: Testen Ihrer Implementation

Fügen Sie wie in Übungsblatt 0 mit `System.out.println` temporär, nur zum Zwecke der Codeentwicklung, ein paar Konsolenausgaben ein, mit denen Sie die die Zeilen- und Spaltenzahl der `World`, alle Zufallswerte unmittelbar nach ihrer Generierung sowie nach Einrichtung der Roboter die Koordinaten der Roboter, ihre Richtungen und ihre Anzahl Münzen ausgeben.

Hinweis:

Die Koordinaten, die Richtung und die Anzahl Münzen eines Roboters `robby` können Sie sehr einfach ausgeben, indem Sie `robby.getX()` (analog `robby.getY()`, `getDirection().name()` und `getNumberOfCoins`) als Parameter in die runden Klammern nach `System.out.print(ln)` schreiben. Kompilieren Sie das Programm so schon einmal und lassen Sie es mehrfach laufen. Prüfen Sie, ob die Konsolenausgaben Ihrer Erwartung entsprechen, und falls nicht, nutzen Sie die Konsolenausgaben zur Fehlersuche. Sobald Sie bei mehreren Programmläufen hintereinander gesehen haben, dass alles stimmt, nehmen Sie die Konsolenausgaben wieder heraus und machen mit dem Folgenden weiter. Es bietet sich an, die Konsolenausgaben nicht sofort ganz zu entfernen, sondern erst einmal mit `//` so auszukommentieren, dass Sie sie bei Bedarf jederzeit einfach wieder einkommentieren können, statt sie noch einmal schreiben zu müssen.

H2: Die Hauptschleife — 15 Punkte

In Klasse `Checkers`, genauer in Methode `runGame` finden Sie die bereits in der Einleitung erwähnte Hauptschleife.

Die Hauptschleife nutzt als Fortsetzungsbedingung den Rückgabewert der Methode `isRunning`, welcher genau dann `true` ist, wenn keiner der beiden Teams gewonnen hat.

Im Rumpf der Hauptschleife sehen Sie bereits die Aufrufe dreier Methoden: Wie die Namen der Methoden schon andeuten, führt Methode `doBlackTeamActions` die Aktionen der schwarzen Steinen und Methode `doWhiteTeamActions` die Aktionen der weißen Steine aus. Methode `updateGameState` überprüft, ob eines der beiden Teams gewonnen hat und ändert gegebenenfalls den Status des Spiels so ab, dass Methode `isRunning` im darauffolgenden Durchlauf `false` liefert und das Spiel terminiert.

In den folgenden drei Teilaufgaben füllen Sie die Rümpfe dieser drei Methoden mit Anweisungen – und damit ist die Hauptschleife dann vollständig.

Hinweis:

Sofern Sie die dritte Methode noch nicht umgesetzt haben, kann Ihr Spiel *zum Ende* mit einer Fehlermeldung abbrechen. Wieso ist das der Fall? Auf diese Frage sollten Sie noch einmal zurückkommen, wenn Ihre Implementation von H2.3 fertig ist.

H2.1: Aktionen der schwarzen Steine — 6 Punkte

Implementieren Sie in Methode `doBlackTeamActions` das Verhalten der schwarzen Steine.

In jedem Durchlauf der Hauptschleife soll ein schwarzer Stein ausgewählt werden, der mindestens eine Münze besitzt und nicht geschlagen wurde. Dieser Stein legt zunächst eine Münze auf das Feld ab, auf dem er gerade steht.

Anmerkung:

Sie dürfen für Ihre Lösung in Methode `doBlackTeamAction` davon ausgehen, dass mindestens ein schwarzer Stein existiert, der nicht geschlagen wurde und Münzen besitzt. Tatsächlich wird dies jedoch erst mit Umsetzung von H2.3 garantiert. Ihr Spiel terminiert bis dahin voraussichtlich nicht korrekt.

Danach bewegt sich dieser Stein auf das erste der vier folgenden möglichen Felder, das innerhalb der Welt liegt und nicht vom weißen Stein besetzt ist:

- Zielfeld 1: oben rechts in Blickrichtung des Roboters
- Zielfeld 2: oben links in Blickrichtung des Roboters
- Zielfeld 3: unten links in Blickrichtung des Roboters
- Zielfeld 4: unten rechts in Blickrichtung des Roboters

Sollte kein Zielfeld existieren, das innerhalb der Welt liegt und nicht vom weißen Stein besetzt ist, wird der Stein ausgeschaltet und gilt damit als geschlagen.

In welche Richtung ein schwarzer Stein nach seinem Zug blickt ist Ihnen überlassen.

Verbindliche Anforderung:

Um den schwarzen Stein zu bewegen, dürfen *nur* die Methoden `move` und `turnLeft` ausgeführt werden.

Beispiele:

Wenn sich der ausgewählte Roboter an Position $(4, 2)$, Blickrichtung `UP` hat und Zielfeld 3 ausgewählt wird, befindet sich der Roboter nach seinem Zug an Position $(3, 1)$. Mit Blickrichtung `RIGHT` würde sich der Roboter nach seinem Zug stattdessen an Position $(3, 3)$ befinden.

Hinweis:

Vermeiden Sie es, die gleichen Aktionen mehrfach – also für jeden schwarzen Stein einzeln – zu implementieren. Beispielweise können Sie eine Variable `chosenStone` vom Typ `Robot` deklarieren. Für diese richten Sie aber *nicht* wie üblich ein neues `Robot`-Objekt ein, sondern lassen Sie mit dem Zuweisungsoperator `=` `chosenBlackStone` auf den gewählten Roboter `blackStone0`, ..., `blackStone4` verweisen. Nutzen Sie in Ihrer Implementation dann `chosenStone` anstatt der jeweiligen anderen Referenz.

H2.2: Aktionen des weißen Steins — 6 Punkte

Implementieren Sie in Methode `doWhiteTeamActions` die Aktion des weißen Steins,

Einen Zug, in welchem der weiße Stein einen schwarzen Stein schlägt, bezeichnen wir als *Schlagzug*. Ein Schlagzug des weißen Steins wird in Methode `doWhiteTeamActions` genau dann ausgeführt, wenn die Möglichkeit hierzu besteht. Andernfalls tut der weiße Stein nichts. Damit der weiße Stein einen schwarzen Stein schlagen kann, muss mindestens ein *Schlagweg* existieren.

Ein Schlagweg ist eine *Diagonale*, die mit dem Feld des weißen Steins beginnt und beliebig viele weitere Felder der Welt enthält, wobei sich je zwei aufeinanderfolgende Felder an einem gemeinsamen Eckpunkt berühren und alle gemeinsamen Eckpunkte auf einer Linie liegen müssen. Das Feld, auf welchem sich der weiße Stein befindet, wird als *nulltes Feld* der Diagonalen bezeichnet – das darauffolgende als *erstes Feld* und so weiter. Damit eine solche Diagonale als Schlagweg zählt, dürfen sich *nur* auf dem vorletzten Feld der Diagonalen ein oder mehrere nicht-geschlagene schwarze Steine befinden. Auf den anderen Feldern der Diagonale dürfen sich auch geschlagene schwarze Steine befinden.

Demnach existieren vom Feld des weißen Steins aus gesehen bis zu vier Richtungen, in die Schlagzüge möglich sein können. Für jede dieser Richtungen kommen unterschiedlich lange Diagonalen für einen Schlagzug in Betracht, wovon jedoch maximal eine für einen Schlagzug genutzt werden kann. Sind in einer Runde mehrere Schlagzüge möglich, wird nur ein Schlagzug ausgeführt. Ihnen steht frei, welcher Schlagzug ausgeführt wird.

Mit einem Schlagzug des weißen Steins wird ein beliebiger sich auf dem Schlagweg befindlicher schwarzer Stein ausgeschaltet (Methode `turnOff` aus Klasse `Robot`), womit dieser schwarze Stein als *geschlagen* gilt. Nachdem ein Schlagzug ausgeführt wurde, befindet sich der weiße Stein auf dem letzten Feld des Schlagwegs.

Verbindliche Anforderung:

Die Bewegungen des weißen Steins dürfen *nur* mit den Methoden `setX` und `setY` ausgeführt werden.

H2.3: Beendigung der Hauptschleife — 3 Punkte

In Klasse `Checkers` finden Sie ein Objektattribut `gameState` vom Typ `GameState`: `GameState` ist eine Enumeration von Zuständen des Spiels, `gameState` der aktuelle Zustand des Spiels. Die in H2 beschriebene Methode `isRunning` liefert genau dann `true`, wenn `gameState` gleich `RUNNING` ist. Weiter kann `gameState` die Konstante `WHITE_WIN` und `BLACK_WIN` annehmen, welche zugewiesen wird, wenn das weiße Team bzw. das schwarze Team gewonnen hat.

Implementieren Sie zum Schluss die Methode `updateGameState`, die den Wert des Attributs `gameState` aktualisiert, wenn einer der folgenden Fälle eingetreten ist.

- Fall 1: Alle schwarzen Steine sind ausgeschaltet und damit vom weißen Stein geschlagen worden. In diesem Fall hat die weiße Partei gewonnen.
- Fall 2: Es existiert mindestens ein ungeschlagener schwarzer Stein und alle ungeschlagenen schwarzen Steine haben all ihre Münzen abgelegt. In diesem Fall hat die schwarze Partei gewonnen.