Prof. Dr. Ralf Hinze
Alexander Dinges, M.Sc.
Sebastian Schloßer, M.Sc.

RPTU
Fachbereich Informatik
AG Programmiersprachen

Übungsblatt 2: Programmierpraktikum 2023

Ausgabe: 02.05.2023
Abgabe: 10.05.2023, 23:59 Uhr

Aufgabe 1 Geometrische Formen (18 Punkte)

In dieser Aufgabe betrachten wir geometrische Formen. Wir definieren dazu eine Schnittstelle `Shape`:

```
public interface Shape {
    boolean contains(int x, int y);
    int rightmost();
    int topmost();
}
```

Die Methode `contains` erwartet als Argument die x- und y-Koordinate eines Punktes und prüft, ob dieser Punkt in der von der Form bedeckten Fläche enthalten ist. Die Formflächen beinhalten jeweils auch die Kanten und Eckpunkte.

Die Methoden `rightmost` und `topmost` bestimmen die größte x- bzw. y-Koordinate der Form.

Beachten Sie, dass wir hier mit dem Typ **`int`** arbeiten, d.h. es sind auch negative Koordinaten möglich.

In dieser Aufgabe müssen Sie einige Klassen erstellen. Klicken Sie dazu in IntelliJ IDEA in der Projektansicht mit der rechten Maustaste auf den `src` Ordner und wählen *New* → *Java Class*. Geben Sie dann den gewünschten Namen der Klasse ein und bestätigen Sie mit Enter.

a) Erstellen Sie eine Klasse `Square`, die das Interface `Shape` implementiert. Die Klasse soll Quadrate repräsentieren. Quadrate haben ihre untere linke Ecke immer am Ursprung des Koordinatensystems, d.h. an der Koordinate $(0, 0)$. Definiert werden Quadrate durch ihre Kantenlänge. Das Quadrat mit der Kantenlänge 0 beinhaltet nur den Punkt $(0, 0)$. Das Quadrat mit der Kantenlänge 2 beinhaltet die Punkte $(0, 0)$, $(0, 1)$, $(0, 2)$, $(1, 0)$, $(1, 1)$, $(1, 2)$, $(2, 0)$, $(2, 1)$ und $(2, 2)$.

   Laden Sie zur Abgabe dieses Aufgabenteils die Datei `Square.java` im Exclaim System hoch.

b) Erstellen Sie eine Klasse `Rectangle`, die das Interface `Shape` implementiert. Die Klasse soll Rechtecke repräsentieren. Rechtecke haben ihre untere linke Ecke immer am Ursprung des Koordinatensystems, d.h. an der Koordinate $(0, 0)$. Definiert werden Rechtecke durch ihre Kantenlängen. Das Rechteck mit der Breite 1 und Höhe 2 beinhaltet die Punkte $(0, 0)$, $(0, 1)$, $(0, 2)$, $(1, 0)$, $(1, 1)$ und $(1, 2)$.

   Laden Sie zur Abgabe dieses Aufgabenteils die Datei `Rectangle.java` im Exclaim System hoch.

c) Erstellen Sie eine Klasse `Union`, die das Interface `Shape` implementiert. Die Klasse soll die Vereinigung zweier Formen repräsentieren. Ein Punkt ist in der Vereinigungsform enthalten, wenn er in mindestens einer der beiden Formen enthalten ist. Definiert wird die Vereinigungsform durch die beiden zu vereinigenden Formen.

Tipp: Vielleicht helfen Ihnen die Methoden aus der Klasse `java.lang.Math`.[1]

Freiwillige Bonusaufgabe: Bauen Sie Ihre Klasse `Union` so um, dass nicht nur zwei sondern beliebig viele Formen damit vereinigt werden können. Gerne dürfen Sie dazu auch die Java Standardbibliothek in vollem Umfang nutzen. Abgeben brauchen Sie dann nur diese erweiterte Version.

Laden Sie zur Abgabe dieses Aufgabenteils die Datei `Union.java` im Exclaim System hoch.

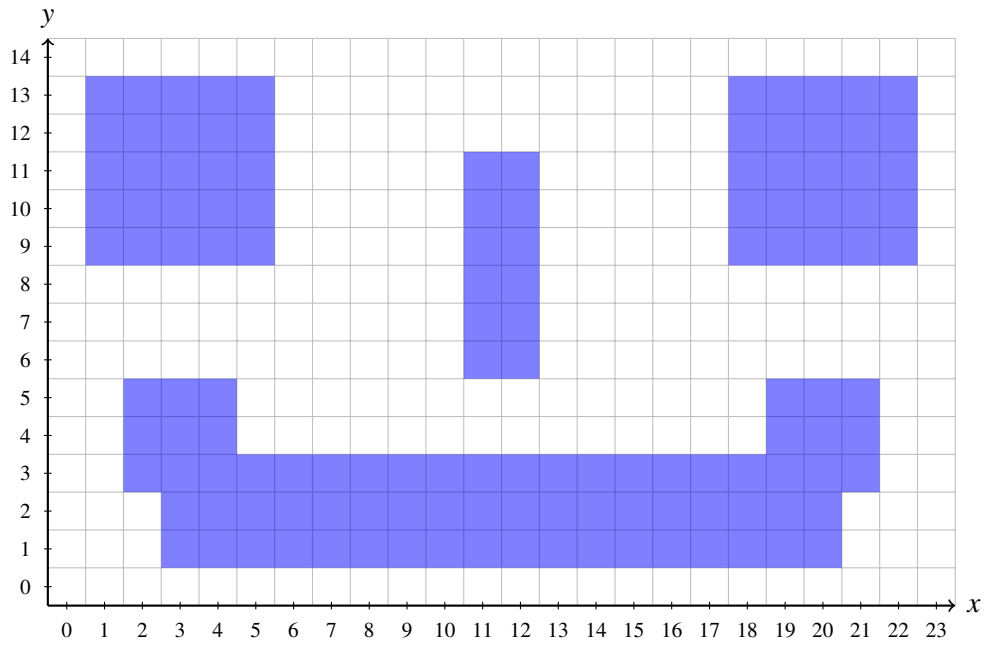
d) Erstellen Sie eine Klasse `Move`, die das Interface `Shape` implementiert. Die Klasse soll die Verschiebung einer Form repräsentieren. Definiert wird die Verschiebung durch die ursprüngliche Form sowie zwei ganze Zahlen, die bestimmen, wie weit die Form auf der x- und y-Achse verschoben wird. Eine positive Verschiebung auf der x-Achse verschiebt die Form nach rechts, eine negative Verschiebung nach links. Eine positive Verschiebung auf der y-Achse verschiebt die Form nach oben, eine negative Verschiebung nach unten.

Hinweis: Die ursprüngliche Form soll hierbei nicht verändert werden. Die `Move` Klasse behält eine Referenz auf die ursprüngliche Form und führt in den drei Methoden Berechnungen durch, die die Verschiebung berücksichtigen.

Laden Sie zur Abgabe dieses Aufgabenteils die Datei `Move.java` im Exclaim System hoch.

e) Implementieren Sie in der Klasse `Smiley` die Methode `Shape getSmiley()`. Die Methode soll die unten blau dargestellte Figur als `Shape` zurückgeben. Verwenden Sie dabei die in den vorherigen Aufgabenteilen erstellten Klassen.

Laden Sie zur Abgabe dieses Aufgabenteils die Datei `Smiley.java` im Exclaim System hoch.



Hinweis: In der Grafik repräsentieren die ausgemalten Flächen die Punkte im Koordinatensystem. Die Nase besteht beispielsweise aus den Punkten (11, 6) bis (11, 11) sowie (12, 6) bis (12, 11).

[1] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Math.html

Aufgabe 2 Vier Gewinnt (18 Punkte)

Hinweis für die Hörer*innen der Vorlesung *Grundlagen der Programmierung* aus dem vergangenen Wintersemester: Sie kennen diese Aufgabe schon von Übungsblatt 12, Aufgabe 2. Gerne können Sie Ihre damalige Lösung zu Rate ziehen und die Unterschiede zwischen F# und Java vergleichen.

In dieser Aufgabe wollen wir den Spieleklassiker *Vier gewinnt*[2] als Kommandozeilenanwendung nachbauen.

Wir generalisieren das Spiel etwas, sodass auch andere Spielfeldgrößen möglich sind und die Anzahl der für einen Gewinn benötigten Steine auf einer Linie variiert werden kann.

Damit wir das Spielfeld als Ascii-Art auf der Konsole ausgeben können, nutzen wir keine Farben sondern markieren die Spieler durch die Zeichen `#` und `*`. Das Spielfeld könnte also wie rechts dargestellt aussehen.
In diesem Beispiel hat das Spielfeld sieben Spalten und sechs Zeilen, also die Standardmaße von *Vier gewinnt*. Wir nummerieren die Spalten von links nach rechts, beginnend bei 0. Die Zeilen werden von unten nach oben nummeriert, ebenfalls bei 0 beginnend. Im Beispiel gehört das Feld in Spalte 2 Zeile 3 also Spieler `*`, das Feld in Spalte 5 Zeile 1 gehört Spieler `#` und das Feld in Spalte 6 Zeile 2 ist noch frei.

```
|0|1|2|3|4|5|6|
| | | | | | | |
| | | | | | | |
|*| |*| | | | |
|#|#|#|*| | | |
|*|#|#|#|*|#|*|
|*|*|#|#|#|*|*|
```

a) Zuerst benötigen wir eine Klasse zur Verwaltung des Spielfelds. Dazu müssen die folgenden beiden Schnittstellen implementiert werden.

```
public interface ReadOnlyGameBoard {
    /**
     * @return the number of columns in the game board
     */
    int getNumColumns();

    /**
     * @return the number of rows in the game board
     */
    int getNumRows();

    /**
     * Get the player who owns the given field.
     *
     * @param column the column (counted from left to right) of the field to check
     * @param row    the row (counted from bottom to top) of the field to check
     * @return the player who owns the field, or null if the field is empty
     */
    PlayerId getField(int column, int row);
}

public interface GameBoard extends ReadOnlyGameBoard {
    /**
     * Put a coin of the given player into the given column.
     *
     * @param column   the column to put the coin into
     * @param playerId the player who should own the coin
     * @throws IllegalMoveException if the column is invalid or full
     */
    void putCoin(int column, PlayerId playerId) throws IllegalMoveException;
}
```

Die beiden Schnittstellen stehen in einer Hierarchie: `ReadOnlyGameBoard` beinhaltet nur Methoden mit lesendem Zugriff auf das Spielfeld, `GameBoard` ergänzt eine Methode, um einen Spielstein in das Feld hineinzuwerfen.

Wir gehen davon aus, dass die Methode `getField` nur mit gültigen Werten aufgerufen wird. In der Methode `putCoin` soll jedoch überprüft werden, ob der Spielzug gültig ist: Einerseits muss die Spaltennummer im gültigen Bereich liegen, andererseits darf die gewählte Spalte noch nicht voll sein. Bei ungültigen Spielzügen wird die Ausnahme `IllegalMoveException` geworfen. Deren Konstruktor erwartet einen `String` mit einer Fehlermeldung.

[2] https://de.wikipedia.org/wiki/Vier_gewinnt

Der Typ `PlayerId` ist ein sogenannter `enum` Typ. Dabei handelt es sich um einen Referenztyp, der die Werte `HASH` (für Spieler `#`), `STAR` (für Spieler `*`) und natürlich **`null`** (von uns benutzt zur Markierung von noch freien Feldern) annehmen kann.

Vervollständigen Sie die Klasse **`class`** `GameBoardImpl` **`implements`** `GameBoard`. Diese Klasse muss also alle vier Methoden implementieren. Der Konstruktor `GameBoardImpl(`**`int`** `columns,` **`int`** `rows)` erhält die Größe des Spielfeldes übergeben. *Tipp:* Sie müssen sich eine Lösung überlegen, um bis zu `columns * rows` verschiedene Einträge vom Typ `PlayerId` zu speichern. Erweitern Sie die Klasse um die dafür benötigte(n) Objektvariable(n) und nutzen Sie diese zur Realisierung der Methoden.

Laden Sie zur Abgabe dieses Aufgabenteils die Datei `GameBoardImpl.java` im Exclaim System hoch.

b) Wir haben Ihnen im Projekt bereits eine Klasse `Game` vorgegeben, die die Spiellogik steuert: Zuerst werden die Konfigurationsparameter (Spielfeldgröße etc.) abgefragt, dann dürfen die Spieler immer abwechselnd ziehen. Dazwischen wird jeweils das Spielfeld auf der Konsole ausgegeben. Sie können das Spiel starten, sobald Sie Aufgabenteil a) gelöst haben.

Probieren Sie aus, ob sich das Spiel richtig verhält: Werden alle eingeworfenen Spielsteine korrekt angezeigt? Fallen die Spielsteine immer bis nach unten durch? Was passiert, wenn Sie versuchen einen Stein in eine volle Spalte oder eine ungültige Spaltennummer einzuwerfen?

Außerdem können Sie die automatischen Tests aus `GameBoardImplTest` ausführen.

Hinweis: Das Spiel beendet sich noch nicht, es geht also selbst dann weiter, wenn ein Spieler gewonnen hat oder alle Felder befüllt sind. Das beheben wir im nächsten Aufgabenteil.

Für diesen Aufgabenteil muss nichts abgegeben werden.

c) Damit das Spiel auch erkennt, wenn ein Spieler gewonnen hat oder ein Unentschieden vorliegt, müssen wir die Spiellogik noch etwas ergänzen. Implementieren Sie in der Klasse `GameOutcomeSolver` die Methode **`static`** `GameOutcome calculateOutcome(...)`. Als Parameter erhalten Sie ein `ReadOnlyGameBoard` sowie einen **`int`** mit der für einen Gewinn benötigten Anzahl an Steinen auf einer Linie. Es ist von uns bewusst so vorgesehen, dass Sie keinen Zugriff auf Ihre tatsächliche Klasse `GameBoardImpl` bekommen. Sie müssen also mit den in der Schnittstelle vorgegebenen Methoden auskommen und können keine Methoden nutzen, die Sie zusätzlich in `GameBoardImpl` ergänzt haben. In größeren Projekten werden einzelne Komponenten oft von unterschiedlichen Teams entwickelt und man hat ebenfalls nur Schnittstellen zur Verfügung, die man selbst nicht erweitern kann.

Der Rückgabetyp `GameOutcome` ist wieder ein `enum` Typ:

```
public enum GameOutcome {
    IN_PROGRESS, WIN_HASH, WIN_STAR, TIE;
}
```

Die Vorlage enthält als Implementierung lediglich **`return`** `GameOutcome.IN_PROGRESS`. Dies zeigt der Spiellogik in der Klasse `Game` an, dass das Spiel noch nicht beendet ist.

Mit **`return`** `GameOutcome.WIN_HASH` gewinnt Spieler `#`, mit **`return`** `GameOutcome.WIN_STAR` gewinnt `*`. Wenn Sie die `PlayerId` des Gewinners haben, dann können Sie auch **`return`** `GameOutcome.winOf(playerId)` benutzen, um sich eine Fallunterscheidung zu sparen.

Mit **`return`** `GameOutcome.TIE` signalisieren Sie ein Unentschieden, also dass alle Felder mit Spielsteinen befüllt sind ohne dass jemand gewonnen hat.

Es reicht aus, wenn Sie auf horizontal und vertikal aneinandergrenzende Spielsteine überprüfen. **Diagonalen brauchen Sie in dieser Teilaufgabe nicht zu betrachten.** Die Testklasse `GameOutcomeSolverTest` enthält automatische Tests für den hier erforderlichen Teil (d.h. ohne diagonale Gewinne).

Da es sich hier um einen etwas umfangreicheren Algorithmus handelt erwarten wir, dass Sie Ihren Code so kommentieren, dass Ihre Lösung einfach nachzuvollziehen ist.

Tipp: Werte eines `enum` Typs (wie zum Beispiel `PlayerId`) können per `==` miteinander verglichen werden. Die Verwendung von `.equals(...)` ist bei `enum`s nicht erforderlich.

Laden Sie zur Abgabe dieses Aufgabenteils die Datei `GameOutcomeSolver.java` im Exclaim System hoch.

d) *Optional:* Erweitern Sie die Spiellogik aus der vorherigen Teilaufgabe um die Erkennung von Gewinnen über diagonal aneinandergrenzende Spielsteine.

Die Testklasse `GameOutcomeSolverOptionalTest` enthält dazu passende automatische Tests.

Laden Sie zur Abgabe dieses Aufgabenteils die Datei `GameOutcomeSolver.java` *im Exclaim System hoch.*

e) *Optional:* Wie Sie beim Ausprobieren des Spiels sicherlich festgestellt haben, können die beiden Spieler wahlweise durch Tastatureingaben oder automatische Spielzüge durch eine KI gesteuert werden. Sie können sogar zwei KI Spieler gegeneinander antreten lassen.

Von künstlicher *Intelligenz* zu sprechen ist bei der Implementierung aus der Vorlage aber deutlich übertrieben. Hier wirft der Computerspieler seinen Spielstein nämlich einfach immer in eine zufällige Spalte ein. Das kann sogar eine volle Spalte sein, in diesem Fall wird der Zug von der Spiellogik verhindert und der Computerspieler wählt eine neue zufällige Spalte aus.

Überarbeiten Sie in der Klasse `ComputerPlayer` die Methode `chooseColumn`. Als Parameter erhalten Sie wie schon in der vorherigen Teilaufgabe ein `ReadOnlyGameBoard` sowie die Anzahl der Steine, die sich auf einer Linie befinden müssen. Außerdem bekommen Sie die `PlayerId` des aktuellen Spielers. Zurückgegeben wird die Spaltennummer, in die der Spieler seinen Spielstein einwerfen soll.

Laden Sie zur Abgabe dieses Aufgabenteils die Datei `ComputerPlayer.java` *im Exclaim System hoch.*