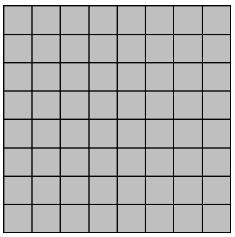
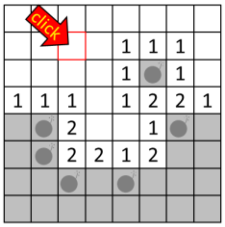
	Minesweeper		AnPr
	Name	Klasse	Datum

1 Situation

Ziel dieser Übung ist es, den Umgang mit zweidimensionalen Arrays zu üben. Als Übungssituation dient das Spiel „Minesweeper“. Die Idee ist, dass in einem zweidimensionalen Feld „Minen“ versteckt wurden, welche von einem Spieler gefunden werden sollen. Haben wir bspw. Ein Spielfeld mit 8x8 einzelnen Feldern, soll der Spieler am Anfang lediglich diese 8x8 Felder sehen, nicht aber, wo die Minen sind. Durch die Auswahl eines beliebigen Feldes wird dieses aufgedeckt. Ist darunter eine Mine, so „explodiert“ sie und das Spiel ist verloren.





Ist keine Mine unter dem gewählten Feld, so decken sich alle zusammenhängenden leeren Felder automatisch auf. Die Grenze wird durch die Felder bestimmt, welche an die Minen angrenzen. Im linken Bild ist das Beispiel gezeigt, in dem oben links ein Klick erfolgt und darauf die zusammenhängenden Felder aufgedeckt werden, die hier weiß dargestellt sind. Zum besseren Verständnis sind die Minen für dieses Bild als Punkte angezeigt – im echten Spiel wären diese natürlich nicht sichtbar. Weiterhin werden bei den freien Feldern die Anzahl der angrenzenden Minen angezeigt.

2 Vorüberlegungen zu Datentypen

Der erste Schritt bei der Erstellung von Algorithmen ist immer, sich Gedanken über die notwendigen Variablen und deren Datentypen zu machen. Bei dieser Aufgabe sticht das zweidimensionale Array für das Spielfeld heraus. Pro Feld gibt es nun mehrere Informationen, welche gehalten werden müssen:

- Mine grenzt an das Feld, das Feld ist aber verdeckt.
- Mine grenzt an das Feld, das Feld ist sichtbar.
- Mine vorhanden, aber verdeckt.
- Mine vorhanden und sichtbar.
- Feld ist leer, das Feld ist verdeckt.
- Feld ist leer, das Feld ist sichtbar.

Wie wir sehen, gibt es drei grundlegende Fälle, jeweils im Zustand „verdeckt“ und „sichtbar“. Für die Abbildung dieser Situationen bieten sich Integer-Zahlen an, welche die beiden Zustände „verdeckt“ und „sichtbar“ mit negativen und positiven Zahlen darstellen. Weiterhin ist es sinnvoll für die angrenzenden Felder die Anzahl der benachbarten Minen als Zahl einzutragen. Überlegen Sie sich, wie viele Minen maximal an ein Feld angrenzen können:

Somit können wir mögliche Werte für ein Feld festlegen:

Fall:	**Wert für „verdeckt“**	**Wert für „sichtbar“**
Feld grenzt an Mine	-1 bis -8	1 bis 8
Mine vorhanden	-9	9
Feld ist leer	0	10

Bei den leeren Feldern gehen wir einen „Sonderweg“, es wird nicht einfach negiert. Der Standardzustand bei der Erzeugung des Arrays ist verdeckt und ohne Mine. Daher nutzen wir hierfür den Initialisierungswert 0. Da „leer“ auch im Zustand „sichtbar“ auftritt und die -0 bei `int` nicht existiert, müssen wir hier einen neuen Wert definieren, nämlich 10. Für die Werte -9, 9, 0 und 10 bieten sich weiterhin Konstanten an, um den Code lesbarer zu machen (bspw. `EMPTY_HIDE = 0;`). Nun können wir den Datentyp für das Spielfeld festlegen:

3 Zählen der Nachbarn

Nun wollen wir den Zugriff auf die Nachbarfelder realisieren. Jedes Feld ist durch seine X/Y – Koordinaten festgelegt und hat somit maximal 8 Nachbarn (bzw. an den Rändern sind es 5 und an den Ecken nur 3). Nun wollen wir nur die Position eines Feldes in einem zweidimensionalen Bereich bestimmen. Hierfür brauchen wir die X- und Y-Koordinate, welche wir in einem einfachen Array ablegen können. Hier bspw. die Koordinaten 5 / 8:

```
int[] pos = {5, 8};
```

Dies ist im Wesentlichen ein Vektor. Auch hier sollten wir mit Konstanten arbeiten:

```
final int X = 0;
final int Y = 1;
```

wodurch bspw. die X-Koordinate mit `pos[X]` ausgelesen werden kann. Die Position der Nachbarn kann man durch die Addition oder Subtraktion von 1 (oder 0) erreichen. Gehen wir davon aus, dass im Spielfeld das Feld links oben die Koordinaten 0/0 besitzt, so würde von einer beliebigen Position X/Y das Feld rechts daneben durch X+1 und Y+0 erreicht werden. Die acht möglichen Richtungen können wir nach den Himmelsrichtungen benennen (North, NorthEast, East, SouthEast, South, SouthWest, West und NorthWest).

X-1/ Y-1	X+0/ Y-1	X+1/ Y-1
X-1/ Y+0	X/ Y	X+1/ Y+0
X-1/ Y+1	X+0/ Y+1	X+1/ Y+1

Dadurch können wir Bewegungsvektoren für die 8 Richtungen festlegen. Hier am Beispiel NorthEast:

```
final int[] NE = {+1, -1};
```

Die einzelnen Bewegungsvektoren können wir nun zu einer Gruppe von Vektoren zusammenfassen. Wollen wir bspw. alle 8 Richtungen für einen Algorithmus prüfen, so legen wir diese in einem weiteren Array fest:

```
final int[][] DIR  = {N, NE, E, SE, S, SW, W, NW};
```

und gehen dann für die Prüfung alle 8 Richtungen von einem gegebenen Punkt durch.

Aufgabe:
Erstellen Sie ein Array mit der rechtsstehenden Struktur – also für alle Felder eines hypothetischen Spielfeldes. Realisieren Sie danach folgende Funktion:

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

```
public int countNeighbours(int[][] data, int[] pos)
```

`data` ist das nebenstehende Array. `pos` gibt an, von welchem Feld die Zahlen der Nachbarfelder zusammenaddiert werden sollen. Dies ist zugleich der Rückgabewert. Hierfür erstellen Sie eine weitere Hilfsfunktion, welche die Position in einem Vektor um eine gegebene Richtung verändert:

```
public boolean move(int[] pos, int[] dir, int maxX, int maxY)
```

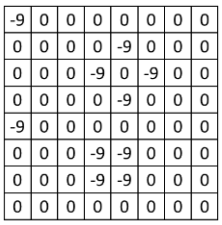
`pos` ist die gegebene Position, welche um `dir` verändert werden soll (wobei `dir` den Richtungsvektoren von oben entspricht, bspw. `NE`). `maxX` und `maxY` geben an, wie groß das Feld ist, in dem sich `pos` bewegt. Für unser Beispiel wäre das jeweils 8. Der Rückgabewert ist `true`, wenn sich `pos` tatsächlich verändert hat und `false`, wenn die Bewegung über den Spielfeldrand hinausgegangen wäre und somit `pos` gleichgeblieben ist.

Testen Sie das Programm mit folgenden Positionen:

0/0	7/7	0/3	3/0	7/3	4/5

Somit haben wir sichergestellt, dass wir mit unserem Algorithmus auf Nachbarfelder zugreifen können. Das Zählen war hierbei lediglich eine Hilfsmethode, um den Test zu ermöglichen.

4 Festlegung Minennachbarn



Nun legen wir die Anzahl der Minennachbarn für die freien Felder fest. Hierzu gehen wir nun anders vor. Wir gehen im Array alle Felder durch und suchen jede verdeckte Mine, die nebenstehend bereits mit -9 eingetragen wurden. Danach gehen wir auf jeden Nachbarn, der keine Mine ist und addieren -1 hinzu (wir bleiben gleich im negativen, da die verdeckten freien Felder ja im Zahlenbereich <= 0 liegen). Die Reduktion der Werte wird in einer eigenen Funktion erledigt:

```
void decreaseNeighbours(int[][] data, int[] pos)
```

Diese wird von einer weiteren Funktion aufgerufen, welche die Minen sucht und die entsprechende Position als Parameter übergibt:

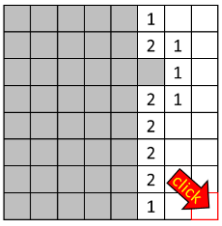
```
void setMineNeighbours(int[][] data)
```

Erstellen Sie die Funktionalität und tragen Sie anschließend die von Ihrem Algorithmus ausgerechneten Werte in die untenstehende Tabelle.

Hinweis: Für den Test benötigen Sie eine Funktion, welche das Array auf der Konsole übersichtlich ausgibt.

-9		0				0	0
				-9			0
0	0		-9		-9		0
				-9			0
-9						0	0
			-9	-9		0	0
0	0		-9	-9		0	0
0	0					0	0

5 Aufdecken der freien Felder



Wenn wir in der oberen Konstellation (für den Fall, dass noch kein Feld aufgedeckt wurde) das Feld unten rechts (also Koordinaten 7/7) aufdecken würden, so müsste der User danach folgendes Bild sehen.

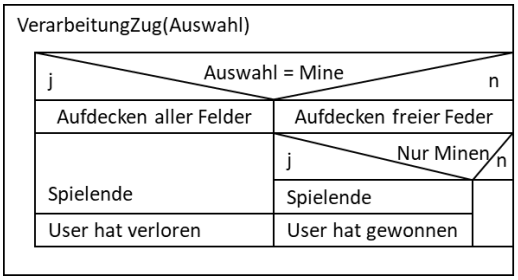
Es müssen also vom Feld 7/7 ausgehend alle Nachbarn aufgedeckt werden – sofern sie keine Mine sind. Ist ein aufgedecktes Feld leer (also Wert = 10), so müssen dort auch wieder alle Nachbarn aufgedeckt werden, sofern sie keine Mine sind – usw.

Erstellen Sie eine Funktion, welche diesen Algorithmus umsetzt. Diese soll später von der Funktion aufgerufen werden, welche den eigentlichen Klick verarbeitet:

```
void showEmpty(int[][] data, int[] pos)
```

Hinweis: Solche Algorithmen setzt man am besten rekursiv um. Achten Sie weiterhin darauf, dass auch auf der initialen Position „`pos`“ das Feld gesetzt werden muss (in unserem Fall bei 7/7 auf „Feld leer, sichtbar“.

6 Prüfung auf Spielstand



Das Spiel kennt drei Zustände:

- Spiel läuft
- User hat verloren
- User hat gewonnen

Zustandsänderungen sind nur dann möglich, wenn der User ein Feld wählt. Insofern benötigen wir eine Funktion, welche uns diese Prüfung übernimmt.

```
int uncover(int[][] data, int[] pos)
```

Für die Rückgabe können wir nun die drei Zustände als konstante Zahlen festlegen. Eine weitere hilfreiche Funktion ist die Prüfung, ob ausschließlich Minen verdeckt sind. Dies bedeutet im Umkehrschluss, dass alle freien Felder aufgedeckt sind und somit Minen gefunden wurden.

```
boolean onlyMines(int[][] data)
```

Schließlich ist es noch hilfreich eine Funktion umzusetzen, welche alle Felder aufdeckt, nachdem der User auf eine Mine gestoßen und das Spiel beendet ist.

```
void showAllFields(int[][] data)
```

Setzen Sie nun die drei Funktionen um.

7 Gesamtspiel

Für das gesamte Spiel fehlt nun nur noch die Eingabe und die zufällige Verteilung der Minen. Da wir nur die Logik im Fokus haben, setzen wir das Programm über die Konsole um. Durch die saubere Trennung von Logik und Ausgabe ist es danach relativ einfach, eine grafische Benutzeroberfläche zu ergänzen. Beginnen wir nun mit der Verteilung der Minen. Erstellen Sie danach eine Funktion, welche zufällige Minen setzt. Diese soll drei Parameter haben:

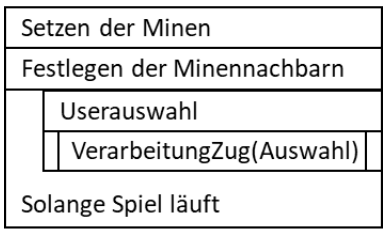
- Feldgröße in Y-Richtung
- Feldgröße in X-Richtung
- Anzahl der Minen

```
int[][] generateField(int xSize, int ySize, int mines)
```

Die Userauswahl soll mit einer Texteingabe erfolgen, wobei X und Y über ein Komma getrennt werden. Die Auswahl für das Feld X = 1 und Y = 2 würde mit der Eingabe `1,2` erfolgen. Hierfür benötigen wir eine Hilfsfunktion für das Parsen der Eingabe. Für die Übung verzichten wir auf das Errorhandling, da in der Praxis das Spiel ohnehin grafisch realisiert werden würde und somit diese Funktion nur für die Entwicklung benötigt wird.

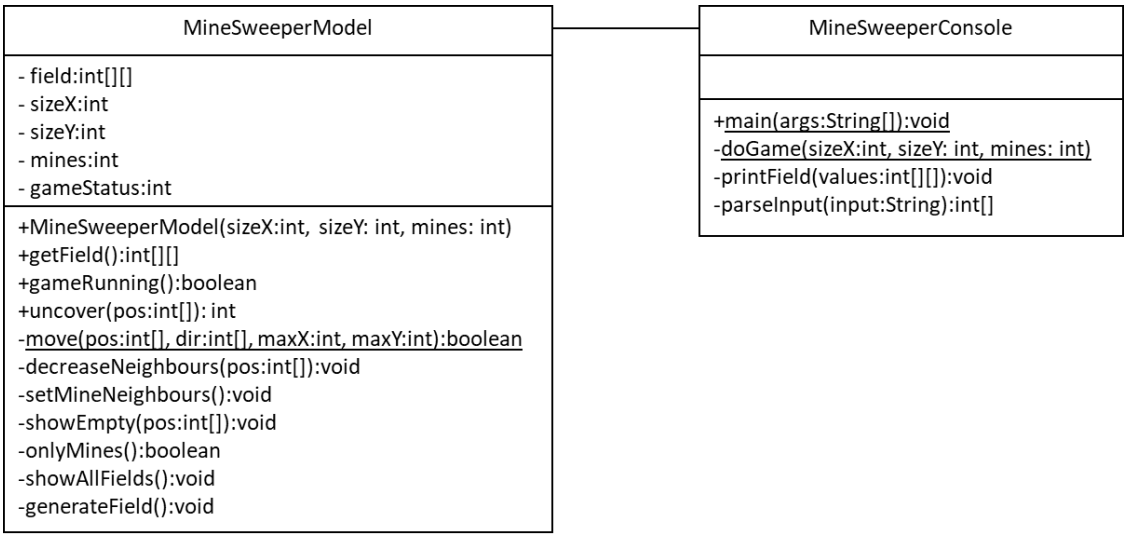
```
int[] parseInput(String input)
```

Der Ablauf des Spiels würde entsprechend des Struktogramms umzusetzen sein.



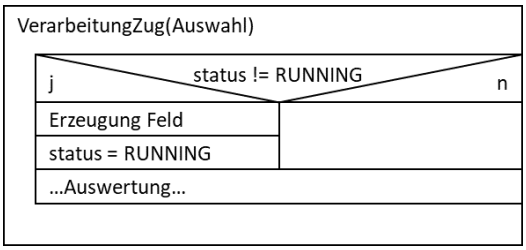
8 Objektorientierte Lösung

Die bisherige Lösung ist rein prozedural. Dies hat im Test zwar Vorteile, bei der Umsetzung und Flexibilität des Ergebnisses, bspw. bei der Implementierung einer GUI Lösung, jedoch Nachteile. Insofern wollen wir eine Klasse „`MineSweeperModel`“ realisieren, welche die gesamte Logik beinhaltet und eine Klasse „`MineSweeperConsole`“ mit dem Spielablauf.



Achten Sie darauf, dass gewisse Informationen nun als Instanzvariablen existieren sollten. Somit können wir auf die Übergabe oder temporäre Berechnung dieser Werte in den einzelnen Methoden verzichten. Bspw. benötigen wir in `uncover` den Parameter des Feldes nicht mehr.

Ein weiteres Problem, auf das man eventuell stößt ist, dass man als ersten Aufdeckversuch zufällig gleich auf eine Mine stößt. Das würde zwar der Realität auf einem echten Minenfeld entsprechen, für das „Spielerlebnis“ ist es jedoch frustrierend. Insofern passen wir die `uncover` Methode so an, dass wir zuerst prüfen, ob das Spiel noch nicht gestartet wurde (also der erste Zug durchgeführt wird). Wenn dem so ist, dann wird zuerst das Spielfeld aufgebaut und dann der Zug ausgewertet:



Somit können wir die Methode um einen Parameter ergänzen, welcher die erste Aufdeckposition beinhaltet. Dieses Feld darf somit vom Zufallsalgorithmus nicht belegt werden:

```
int[][] generateField(int[] block)
```