Grundlagen der Programmierung

Marcel Lüthi
Andreas Morel-Forster
HS 23

Universität Basel
Fachbereich Informatik

Übung 9

Voraussetzung

- Ein JDK ist installiert.
- Installierte IDE, Visual Studio Code sowie die Plugins für Java und Gradle
- Wenn Sie die Vorlesung verpasst haben, dann empfehlen wir Ihnen die Unterlagen anzuschauen.
- Die Zip-Datei, die auch dieses Übungsblatt enthält, muss entpackt werden. Es enthält die gesamte Übungsumgebung. Schreiben Sie ihre Lösungen in die dafür vorgesehenen Dateien, wie in der jeweiligen Übungsaufgabe angegeben.

Wichtiger Hinweis

- *Achten Sie auf guten Programmierstil.*

Hinweise zum Kompilieren und Ausführen der Programme In dieser Übung ist der Code das erste mal in Paketen organisiert. Dies müssen Sie beim Kompilieren und Ausführen berücksichtigen. Um beispielsweise die Klasse Fraction im Paket fraction zu kompilieren, wechseln Sie wie gewohnt in das Verzeichnis `src/main/java`, geben dann aber beim Kompilieren den Pfad zur Datei mit an:

```
> javac fraction/Fraction.java
```

Beim Ausführen müssen Sie dann entsprechend den gesamten Namen, inklusive Paket angeben:

```
> java fraction.Fraction
```

Aufgabe 9.1 (Fraction, 3 Punkte)

Sie finden im Verzeichnis src/main/java/fraction die Klassen Fraction und ReducedFraction. Implementieren Sie die Methode reduce in der Klasse Fraction, die einen Bruch kürzt. Die Klasse ReducedFraction soll nun alle Methoden von Fraction anbieten, aber es soll immer ein gekürzter Bruch resultieren. Nutzen Sie dabei die bereits vorhandene Funktionalität in der Klasse Fraction, aber ohne diese zu kopieren.

Aufgabe 9.2 (Game of Life)

Das Spiel des Lebens (Game of Life) geht auf den Mathematiker John Conway zurück und funktioniert nach folgenden Prinzipien:
`( http://www.math.com/students/wonders/life/life.html )`
Die Grundeinheit sind Zellen, die in einer Matrix angeordnet sind. Jede Zelle kann lebendig oder tot sein. Jede Zelle hat acht Nachbarn, wobei Randzellen die Zellen des gegenüberliegenden Randes als Nachbarn haben. Der Zustand der Zellen (lebendig oder tot) ändert sich von Generation zu Generation. Die aktuelle Zellpopulation beeinflusst die darauf folgende Generation nach folgenden Regeln:

(a) Eine tote Zelle mit genau drei lebenden Nachbarn erwacht zum Leben (birth).

(b) Eine lebende Zelle mit zwei oder drei lebenden Nachbarn bleibt am Leben (survival).

(c) Alle anderen lebenden Zellen sterben (overcrowding or loneliness).

Sie finden im Verzeichnis `src/main/java/` verschiedene Interface und abstrakte Klassen. Sie sollen diese nacheinander vervollständigen und benutzen. Lesen Sie dazu immer auch die Kommentare im Quellcode.
Wenn Sie alles bearbeitet haben, können Sie Ihren Code mit den automatisierten Tests testen. Diese finden Sie im Verzeichnis `src/main/test/java`. Die Tests sind alle auskommentiert, da es ansonsten während der Entwicklung zu Kompilationsfehlern kommen würde. Kommentieren Sie diese wieder ein und testen Sie Ihr Programm.

Als erstes wollen wir einige Methoden in den Interfaces und den abstrakten Klassen implementieren.

Field Schauen Sie sich als erstes die Datei `Field.java` an. Darin ist ein Interface `Field` definiert. Bis auf eine *default* Methode, sind alle Methoden *abstract*. Vervollständigen Sie die *default* Methode `print` welche das Feld ausgibt. Verwenden Sie in der Implementation der `print` Methode nur Methoden dieses Interfaces.
Tote Zellen sollen als '.' und lebende Zellen als '@' ausgegeben werden. Die Zellen einer Zeile (row) sollen auch bei der Ausgabe auf eine Zeile geschrieben werden.

SaveAccess Schauen Sie sich nun die Datei `SaveAccess.java` an. Hier ist ein Interface definiert, welches eine Methode für den Zugriff auf eine Zelle deklariert. Der Zugriff soll geschützt erfolgen, und Zugriffe ausserhalb des gültigen Bereichs sinnvoll behandeln. Noch ist aber keine Implementation gegeben, da es nur ein Interface ist. Sie müssen in diesem Interface nichts implementieren.

Rules Schauen Sie sich als nächstes die Datei `Rules.java` an. Diese Abstrakte Klasse definiert die Regeln für das Game-of-Life und leitet vom Interface `SaveAccess` ab. Implementieren Sie die generellen Methoden `countNeighbours` sowie `willBeAlive` mit Hilfe der `get` Methode von `SaveAccess`. Wie Sie sehen, können Sie die Methoden implementieren in dem Sie das Interface benutzen, obwohl noch keine konkrete Implementierung der Methode `get` vorhanden ist. Somit definiert auch diese Klasse noch nicht, wie man mit Zugriffen ausserhalb des Feldes umgeht.

GameOfLife Als nächstes sollen Sie nun in der Datei `GameOfLife.java` die abstrakte Klasse vervollständigen. Implementieren Sie die Methoden `evolve` und `print`, sowie den Konstruktor. Dabei können Sie die abstrakten Methoden der Klasse benutzen. Die Methode `evolve` soll dabei zuerst die nächste Generation in einem zweiten Feld speichern. Erst wenn dieses Feld gefüllt ist, soll das alte Feld durch das neue ersetzt werden.

Nun wollen wir zu konkreten Implementation kommen.

BooleanField Implementieren Sie eine Klasse BooleanField in einer neuen Datei, welche das Interface Klasse `Field` implementiert. Schreiben Sie einen Konstruktor, welcher ein 2d-Array von `boolean` erstellt, entsprechend einer übergebenen Höhe und Breite. Dabei soll die Klasse nicht abstakt sein und somit für alle noch nicht implementierten Methoden eine Implementation definieren.

CircularBoundaryRule Erstellen Sie eine neue Java-Klasse `CircularBoundaryRule` in einer eigenen Datei. Die Klasse soll die abstrakte Klasse `Rules` erweitern. Definieren Sie nun den zirkulären Zugriff auf Zellen. Das heisst, dass für Zellen, welche ausserhalb des gültigen Bereichs liegen, auf Zellen auf der anderen Seite zugegriffen werden. Implementieren Sie dazu die Methode `get`.

MyGameOfLife Erstellen Sie eine Klasse `MyGameOfLife` welche die Klasse `GameOfLife` erweitert. Dabei sollen die drei abstrakten Methoden `createField`, `createRules` sowie `init` implementiert werden. Für die ersten beiden Methoden, verwenden Sie die von Ihnen geschriebene Klassen. In der initialisierungs Methode können Sie jede Zelle zufällig auf lebend oder tot setzen, indem Sie `Math.random()` benützen. Vergleichen Sie dazu den zurück gegebenen Wert, ob er kleiner als ein gewisser Wert ist (z.B. 0.3).

Nun wollen wir noch das Programm schreiben.

Application Vervollständigen Sie nun noch die main-Methode in der Klasse `Application`. Die Methode soll eine Instanz Ihres Game-of-Lifes erstellen und 100 Iterationen ausgeben. Sie können nun Ihr Programm ausführen. Tut es was es soll?

Als Programmierende welche noch nicht so viel Erfahrung haben, kann es sein, dass Sie den Vorteil von Klassenhierarchien nicht sehen. Zugegeben, es wirkt in diesem Fall auch etwas übertrieben. Um den Nutzen etwas zu verdeutlichen, überlegen Sie sich die folgenden Aufgaben oder implementieren Sie diese direkt.

AliveBoundaryRule Schreiben Sie eine Klasse `AliveBoundaryRule`, welche für alle Zellen ausserhalb des gültigen Bereichs `true` zurück gibt. Wo müssen Sie nun überall etwas anpassen, dass die neue Klasse anstelle der alten verwendet wird?

IntegerField Schreiben Sie eine Klasse `IntegerField`, welche ein 2d Array von `int` verwendet um das Feld zu speichern. Wo müssen Sie nun überall etwas anpassen, dass die neue Klasse anstelle der alten verwendet wird?

ZUSATZ: Initialisierungen Wenn Sie mögen, können Sie auch noch alternative Initialisierungen, welche das Game-of-Life mit einem der folgenden Mustern initialisiert, implementieren:

```
                                ......
                    .....       .@.@..
....                ..@..       ..@@..
.@@.                ..@..       ..@...
.@@.                ..@..       ......
....                .....       ......
(Block)             (Blinker)   (Glider)
```

Abgabe Erstellen Sie eine Zip-Datei der gesamten Übungsumgebung (also des Verzeichnisses `uebung009`) und laden Sie dieses auf Adam hoch.