Übungsblatt 11

Musterlösung

Aufgabe 1 Einmal ein Lambda, bitte

Da Listen bereits direkt eine Methode *forEach* anbieten, ist dies einfach:

```
20          level.getActors().forEach(actor -> actor.act());
```

Aufgabe 2 Strömen statt Schleifen

Die Variante *IntStream.iterate* mit drei Parametern in Kombination mit *forEach* entspricht ziemlich genau einer Zählschleife. Deshalb müssen zwei solche Streams ineinander verschachtelt werden, um über das zweidimensionale Feld zu laufen:

```
62          IntStream.iterate(0, y -> y < field.length, y -> y + 2)
63                  .forEach(y -> IntStream.iterate(0, x -> x < field[y].length(), x -> x +
                            2)
64                  .forEach(x -> gameObjects.add(new GameObject(x / 2, y / 2, 0,
                            neighborhoodToFilename[getNeighborhood(x, y)]
65                  .replace("path", getCell(x, y) == 'W' ? "water" : "path")))));
```

Aufgabe 3 In Strömen rechnen

Hier kann ein *IntStream* genommen werden, um über die vier Richtungen zu iterieren. Da in Einerschritten gezählt wird, kann hier statt *iterate* auch das etwas einfachere *range* genutzt werden. Durch *filter* werden nur die Richtungen erhalten, in denen auch eine Verbindung zum Nachbarn besteht. Aufsummiert werden sollen aber nicht die Richtungen selbst, sondern die ihnen entsprechenden Bits. Dies wird durch ein *map* erreicht, das diese Umrechnung für jede noch vorhandene Richtung vornimmt. Im Ergebnis müssen die einzelnen Bits ver-odert werden, was mit *reduce* erreicht wird. Da die Richtungsbits disjunkt sind, kann hier auch – wie in der ursprünglichen Implementierung – die Addition verwendet werden. Somit könnte *reduce* hier auch durch die parameterlose Methode *sum* ersetzt werden, die dasselbe Ergebnis liefert.

```
106         return IntStream.range(0, neighbors.length)
107             .filter(direction -> getCell(x + neighbors[direction][0], y + neighbors[
                        direction][1]) != ' ')
108             .map(direction -> 1 << direction)
109             .reduce(0, (a, b) -> a | b);
```

Aufgabe 4 In Strömen sammeln

Hier reicht es, von einer Vorlesungsfolie abzuschreiben. *toList* liefert alle Elemente des Streams als Liste zurück:

```
52          final List<String> lines;
53          try (final BufferedReader stream = new BufferedReader(new InputStreamReader(Game.
                    Jar.getInputStream(fileName)))) {
54              lines = stream.lines().toList();
55          }
```

Aufgabe 5 Kreativ strömen

Einen wirklichen Ersatz für eine *while*-Schleife bieten Java-Streams eigentlich nicht, denn es wird erwartet, dass immer Daten durch den Strom fließen. Insofern muss dafür gesorgt werden, dass solche Daten erzeugt werden, auch wenn sie eigentlich gar nicht benötigt werden. In der folgenden Implementierung wurde hierfür die Liste der Akteure gewählt, die von *generate* immer wieder durch den Strom geleitet wird, weil diese ja auch wenigstens in der „Schleife“ verwendet wird. Das *takeWhile* sorgt dafür, dass die Verarbeitung abbricht, sobald die Bedingung falsch wird. *forEach* führt, wie üblich, den Schleifeninhalt aus.

```
22          Stream.generate(() -> level.getActors())
23                  .takeWhile(actors -> actors.get(0).isVisible())
24                  .forEach(actors -> actors.forEach(actor -> actor.act()));
```

Alternativ hätte auch *iterate* verwendet werden können, das mit der Identitätsfunktion zum Weiterzählen arbeitet:

```
22          Stream.iterate(level.getActors(), actors -> actors.get(0).isVisible(), actors ->
                    actors)
23                  .forEach(actors -> actors.forEach(actor -> actor.act()));
```