README.md

Bonusaufgabe EidP

Von: Raiden Erdmann,

Studiengang: Wirtschaftsinformatik

Matrikelnr.: 7209573

a

Allgemeines

Für die erste Teilaufgabe war es wichtig, herauszufinden, wie die Buchstaben gezeichnet werden sollen. Dazu habe ich mir überlegt, wie man als Erstklässler die Buchstaben gelernt hat.

Hier werden Symbole Linie für Linie, schrittweise erstellt. Beispielsweise wird eine `E` gezeichnet, indem zuerst der linke vertikale Strich von oben nach unten gezogen wird. Anschließend folgen von oben nach unten die horizentalen Striche, welche von links nach rechts gezogen werden. Hierbei lässt man den mittleren Stich etwas kürzer.

Diese Überlegungen lassen sich in Arbeitsanweisungen umschreiben, welche wiederum sich durch Schleifen relativ einfach reproduzieren lassen.

Rundungen bei dem Buchstaben R

Kniffelig wird es allerdings bei Rundungen. Da einfache Iterationsschleifen von gradliniger Natur sind, Lösung gefunden werden, die Rundung nachzustellen. Das Nächstbeste, wofür ich mich auch entschieden hatte, ist eine zur Rasterung angelehnte Methode, bei der die Kurve durch beliebig viele Polygone, und dazwischen gezogene Linien dargestellt wird. Durch ein zusäzlichen Polygon bekommt der Bauch des `R` 's eine Dach-Form, ähnlich wie das `>`. Ein weiteres Polygon würde zu einem Trape führen usw. Um das Dach zu zeichnen, habe ich verschiedene Methoden probiert.

Eine davon war es mit einer for-Schleife zwei Linien zu zeichnen, welche von der oberen Horizontale und der mittigen des `R` 's ausgehen und diagonal so verlaufen, dass sie sich in der Mitte treffen. Hierbei war ein Problem, dass die Mitte, bei ungeraden Zahlen nicht nicht immer geschlossen war. So gab es eine Lücke, an der Spitze des Daches `>`.

Um eine komplette Rundung zu verwirklichen, habe ich probiert die koordinaten der Kurve über die Mathematische Kreisformel $$c = x^2+y^2$$ zu berechnen. Hier wäre `c = n` der Radius des Halbkreises und y^2 wäre eine Zelle von `char[][]`. Der Ansatz war es die Spalte zu berechnen:

```
import java.math;
// Angenommen n = 20
int n = 20;
char[][] field = new char[n][n];

// c = x^2 + y^2 => x = sqrt(c - y^2)
for(int i = 0; i < n; i++){
    int y = (int) sqrt(n - x^2);
    field[i][y] = '+';
}
```

Da dies auf kleiner Skala klobig und auf größeren eigenartige Proportionen hatte, habe ich mich letztendlich dafür entschieden, eine Diagonale Linie zu ziehen, welche auf halbem Weg gespiegelt wird. Das hat den Vorteil, dass Feinheiten einfach nachgebessert werden können:

```
int n = 20;
char[][] field char[n][n];

// Da das 'R' zu diesem Zeitpunkt aussieht wie ein 'F', kommen die Brüche zustande.
// n/8 ist z.B. der halbe Weg zwischen den Horizontalen des R.
```

```
// n/2 ist aus mathematischer Sicht sowas wie "1-x", also die
// rechte 'Wand' der Box in der sich der linke Buchstabe befinden soll.
// Ab n/2 beginnt der Bereich des rechten Buchstaben
for (int i = 0; i < n / 4; i++) {
    field[i][(i <= n / 8) ? ((n / 4) + i) : ((n / 2) - i)] = '+';
    // In worten:
    // Bin ich bereits zur Hälfte da?
    //  => Gehe einen nach unten und einen nach links
    // sonst:
    // => Ziehe einen nach unten und einen nach rechts
}
```

Das `E` hingegen bestand aus geraden Linien und war somit deutlich einfacher. Von oben nach unten und drei Horizontale Lienen.

Tests

Getestet habe ich viel durch wiederholtes Ausführen mit geänderten Werten. Zur Hilfe habe ich eine `printField` Methode erstellt, die Zeilen und letzte Ziffer der Spalten zusammen mit dem Feld ausgibt. Zusätzlich habe ich bei der Schwierigkeit mit der Rundung des `R` auch den Debugger angewender, um manuell durch die for-Schleife zu iterieren und verrückungen etc. zu korrigieren.

b

Allgemeines

Für das `Game of Life` habe ich die Regeln verwendet die ich am öftesten gefunden habe, bei denen es keine 'leeren' Zellen gibt, sondern nur tote und lebendige. Mir wurde aus der Aufgabenstellung nicht klar, welchen Zweck ein dritter Zustand erfüllen sollte, also habe ich angenommen, dass 'leere' und 'tote' felder, einfach leere felder sind.

Um im allgemeinen Fall zu Prüfen, ob eine Zelle lebendig ist oder nicht, betrachtee ich die Zelle selbst und die in unmittelbarer Umgebung. Im Fall der Zelle (2,3) wird also der Zustand von (2,3) und von allen in (2 ± 1, 3 ± 1) gewertet. Im Anschluss werden folgende Regeln beachtet:

- Eine Zelle bleibt am Leben, wenn Sie 2 / 3 lebendige Nachbarn hat.
- Eine leere Zelle erwacht zum Leben, wenn genau 3 benachbarte Zellen leben
- Alles anderen lebenden Zellen sterben aus, die leeren Zellen bleiben leer

Randbedingungen

Wenn man die Zellen am Rand des Feldes untersucht, muss man davon ausgehen, dass Nachbarsfelder, die nicht im Spiel sind, als leer betrachtet werden. Da Arrays ab Index 0 beginnen, gibt es keine Möglichkeit Zellen außerhalb des Arrays selbst du betrachten. Dies würde in einer `ArrayIndexOutOfBoundsException` resultieren. Wenn man beispielsweise die Zelle (0,0) untersucht, so betrachet man die Nachbarszelle (-1,0) als leer, da man Sie nicht direkt ansprechen kann. Hier habe ich eine try-catch methode angewendet, bei der `ArrayIndexOutOfBoundsException` abgefangen werden und irgnoriert werden:

```
for (int i = x - 1; i < x + 2; i++) {
    for (int j = y - 1; j < y + 2; j++) {
        try {
            if (field[i][j] == '+' && (!(i == x && j == y))) {
                ++liveNeighbours;
            }
        } catch (ArrayIndexOutOfBoundsException e) {
            // assume '-' for outside of border
            // just dont count it
        }
    }
}
```

Hier wird ebenfalls teilweise gezeigt, wie ich den Zustand einer Zelle in der nächsten Generation werte, die zweite Schwierigkeit dieser Aufgabe. Die mir bekannten Regeln kann man wie folgt zusammenfassen: Eine Zelle lebt in der nächsten Generation wenn:

- Genau 3 Nachbarn leben
- Die Zelle selbst und 2 oder 3 Nachbarszellen leben

Es werden zunächst alle Nachbarn einer Zelle auf Lebendigkeit geprüft, die Anzahl der lebenden Nachbarn festgehalten (s.o.).

Anschließend werden die Regeln angewendet:

```
// from public static boolean isCellLive()
return (liveNeighbours == 3 || (liveNeighbours == 2 && field[x][y] == '+')) ? true : false;
```

Dies wird für Jede Zelle des Feldes abgefragt und in einem neuen Feld eingetragen, da sonst der 'Spielstand' soz. verfälschtwerden würde.

Test

Für die Tests habe ich, zu einem die Initialen aus Aufgabe a, vor Allem aber Objekte verdendet, die dafür bekannt sind, dass sie nicht sterben. Beispielsweise eine Reihe von 3 lebendigen Zellen ('+', '+', '+') ist dafür bekannt, dass sie in der nächsten Generation von der Horizontalen, in die Vertikale übergeht, und in die darauf folgende Generation wieder zurück. So oszilliert das Objekt und stribt nicht (ausser ein anderes Objekt kommt in die Quere).

Ein zweites Objekt ist der `Glider`, welcher über das Feld wandert.

C

Für c habe ich die Regeln von `Game of Life` erweitert. Es gibt nun eine dynamische Regel, die das Spiel über die Generationen unterscheidlich stark beinflusst.

Nachdem die bekannten Regeln angewendet wurden, werden die 'äußeren' Zellen ausgewertet:

```
for (int i = x - 2; i < x + 3; i++) {
    try {
        if (field[i][y - 2] == '+' || field[i][y + 2] == '+') {
            ++liveOuterNeighbours;
        }
    } catch (ArrayIndexOutOfBoundsException e) {
        // assume '-' for outside of border
    }
}
for (int i = y - 1; i < y + 2; i++) {
    try {
        if (field[x - 2][i] == '+' || field[x + 2][i] == '+') {
            ++liveOuterNeighbours;
        }
    } catch (ArrayIndexOutOfBoundsException e) {
        // assume '-' for outside of border
    }
}
```

Und anschließend die Neuen Regeln angewendet:

```
if (liveOuterNeighbours + (2 * liveOuterNeighbours) > 10)
    return false;
if (liveOuterNeighbours > 8)
    return true;
```

Die Bedingung, dass nach `n` /2 Generationen noch immer lebendige Zellen existieren, jedoch nach geraumer Zeit nicht mehr, ist jedoch schwierig. Es gibt immer wieder bestimmte `n` bei denen Objekte entstehen, wie z.B. ein Quadrat aus 4 Zellen, welche nach der Urspünglichen Implementierung nicht sterben. Auf der anderen Seite gibt es wieder Ausgangszustände, bei denen nach 4 Generationen das Feld leer ist. Ich habe leider keine Regel gefunden, bei der ausnahmslos für ein beliebiges n, diese Bedingungen eingehalten werden.

Quellen

- Allg. Regeln: https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life
- Regelimplementierung (für Verständnis): https://github.com/MarioTalevski/game-of-life/blob/abe131dd5d212fd2c044fb827137a964c98551ea/GameOfLife.cpp#L230
- Ein wenig Historie: https://www.youtube.com/watch?v=R9Plq-D1gEk&ab_channel=Numberphile

Bestätigung

Hiermit bestätige ich, Raiden Erdmann, dass ich ausschließlich die angegebenen Quellen verwedet habe und ohne Hilfe die Aufgaben bearbeitet habe.