Prof. Dr. Stefan Brass
Dipl.-Inform. Annett Thüring
Dipl.-Inform. Steffen Schiele
Dipl.-Inform. Andreas Bienert

Martin-Luther-Universität Halle-Wittenberg
Institut für Informatik

Wintersemester 2014/15

Objektorientierte Programmierung: Hausaufgabenblatt 7

Abgabe: 11.12.2014, 12:00 Übung: 15./16.12.2014

Hausaufgabe 7: (4 Theoriepunkte)

Ein magisches Quadrat der Kantenlänge n ist eine Matrix Q der Größe $n \times n$, die die Zahlen von 1 bis n^2 jeweils genau einmal enthält, und bei der die Summen der Zeilen, die Summen der Spalten, und die Summen über beiden Diagonalen alle gleich sind. Ein magisches Quadrat der Kantenlänge 3 ist also:

2	7	6
9	5	1
4	3	8

Mehr Informationen zu magischen Quadraten finden Sie z.B. in der Wikipedia:

`http://de.wikipedia.org/wiki/Magisches_Quadrat`

Ziel dieser Hausaufgabe ist ein Programm, das magische Quadrate berechnet, und zwar potentiell begliebiger Kantenlänge n. Für den Anfang reicht ein "Brute Force" Ansatz, indem einfach alle möglichen Quadrate konstruiert werden, und dann auf die Eigenschaft der magischen Quadrate geprüft werden. Praktisch geht ein so einfacher Ansatz nur für die Kantenlänge $n = 3$, dort gibt es ja schon $9^9 = 387\,420\,489$ Quadrate, die zu prüfen sind (wenn man jede der 9 Positionen unabhängig von einander mit den 9 möglichen Werten belegt). Der zu Testzwecken verwendete Rechner hat das in ca. 8s geschafft. Sie dürfen aber (freiwillig) die Generierung verbessern, so dass aussichtslose Kandidaten frühzeitig ausgeschieden werden. Dann schaffen Sie vielleicht auch höhere n.

Die Prüfung der Eigenschaft haben wir für Sie schon programmiert. Ihre Aufgabe ist das Generieren möglicher Kandidaten. Da wir mehrdimensionale Arrays noch nicht hatten (und es die strenggenommen in Java auch gar nicht gibt), benutzen wir ein eindimensionales Array der Größe $n * n$ für die Matrix. Das Matrix-Element $Q(i, j)$ speichern wir im Array in `arr[i*n+j]`, d.h. die Matrixelemente sind zeilenweise hintereinander gespeichert. Für $n = 3$ gilt z.B.

$Q(0,0)$	$Q(0,1)$	$Q(0,2)$	$Q(1,0)$	$Q(1,1)$	$Q(1,2)$	$Q(2,0)$	$Q(2,1)$	$Q(2,2)$
`arr[0]`	`arr[1]`	`arr[2]`	`arr[3]`	`arr[4]`	`arr[5]`	`arr[6]`	`arr[7]`	`arr[8]`

Wenn man alle Möglichkeiten durchprobieren will, den 9 Plätzen im Array die Zahlen von 1–9 zuzuweisen, könnte man das mit 9 ineinander geschachtelten Schleifen machen:


```
for(i = 1; i <= 9; i++) {
        a[0] = i;
        for(j = 1; j <= 9; j++) {
                a[1] = j;
                ...
                        for(q = 1; q <= 9; q++) {
                                a[8] = q;
                                if(isMagicSquare(a))
                                        printSquare(a);
                        }
                ...
        }
}
```

Es macht natürlich keinen Spass, das aufzuschreiben. Außerdem geht es so nur für ein fest vorgegebenes n. Ihre Aufgabe ist, dieses Programmstück in eine rekursive Methode zu überführen. Das vorgegebene Hauptprogramm kann unter folgender Webadresse abgerufen werden:

`http://www.informatik.uni-halle.de/~brass/oop14/homework/Quadrat.java`

Es ist auch unten abgedruckt. In der `main`-Methode finden Sie den Aufruf

`genMagicSquares(n);`

wobei `n` eine ganze Zahl ist (z.B. `3`). Schreiben Sie diese Methode. Sie können folgende Methoden benutzen:

- `static boolean isMagicSquare(int n, int[] square)`:
  Diese Methode testet ein gefülltes Array auf die Eigenschaft, ein magisches Quadrat zu sein. Das Array muss die richtige Größe haben (`n*n`) und darf nur Zahlwerte von `1` bis `n*n` enthalten.

- `static void printSquare(int n, int[] square)`:
  Diese Methode druckt ein Array als Matrix aus. Außerdem zählt die Methode (in einer globalen/statischen Variable) mit, wie häufig sie aufgerufen wurde. Diese Anzahl wird von Hauptprogramm am Ende als Anzahl magischer Quadrate ausgegeben. Um nicht zu lange Ausgaben zu bekommen, werden nur die ersten 20 Quadrate wirklich ausgegeben, danach wird nur noch gezählt. Die Grenze `20` ist als Konstante `MAX_PRINT` definiert und könnte von Ihnen bei Bedarf geändert werden.

Gegenüber dem obigen Beispiel-Programm wurden beide Methoden also um den zusätzlichen Parameter `n` erweitert (jetzt soll es ja im Prinzip für beliebige `n` funktionieren — genügend Rechenzeit vorausgesetzt).

Sie dürfen eigene Methoden (und wenn es unbedingt sein muss, auch globale/statische Variablen) zu der vorgegebenen Klasse hinzu fügen. Stilistisch wäre es gut, diese als `private` zu deklarieren. Z.B. könnte man für die Rekursion eine Methode

`static void genRecursive(int n, int[] square, int i)` definieren, die für die Position `i` im Array `square` zuständig ist: Wenn das Array gefüllt ist, also $i = n^2$ ist, wird der Test auf die "Magisches Quadrat" Eigenschaft aufgerufen, und das Array ggf. ausgedruckt. Ansonsten werden in einer Schleife nacheinander alle möglichen Werte in `square[i]` eingetragen, und der Rest der Aufgabe jeweils durch rekursiven Aufruf mit Parameterwert `i+1` erledigt.

Es sei der Fall für $n = 3$ betrachtet. Sie starten also die Rekursion mit $i = 0$. Zuerst wird `a[0]=1` gesetzt, und die Methode rekursiv aufgerufen (mit $i = 1$). Dann wird `a[0]=2` gesetzt, und die Methode wieder rekursiv aufgerufen (wieder mit $i = 1$). Und so weiter, bis schließlich zum rekursiven Aufruf für `a[0]=9` (wieder mit $i = 1$). Jeder der neun Aufrufe für $i = 1$ macht das Gleiche an der Array-Position 1. Setzt also zuerst `a[1]=1` und ruft sich rekursiv mit $i = 2$ auf. Insgesamt sind das pro Aufruf mit $i = 1$ neun Aufrufe mit $i = 2$. Darunter sind allerdings einige Fälle, die niemals zu einem magischen Quadrat vervollständigt werden können, z.B. wenn in `a[0]` und `a[1]` der gleiche Wert gespeichert ist. Das Programm funktioniert trotzdem, weil am Ende der Rekursion (für $i = 9$) ja die Bedingung geprüft wird. Man hat so aber viel umsonst gerechnet. Es steht Ihnen frei, rekursive Aufrufe zu sparen, wenn klar ist, dass sie nie zu einer Lösung führen können.

Hinweis: Das Hauptprogramm akzeptiert einen Parameter aus der Kommandozeile, also z.B.

```
java Quadrat 3
```

Sie können es aber auch ohne Kommandozeilen-Argumente aufrufen, dann wird ein Wert für n abgefragt.

Noch ein Tipp: Wenn Sie die Rekursion ausprobieren wollen, könnten Sie z.B. $n = 2$ wählen, und den Test auf "Magisches Quadrat" weglassen: Dann werden die 256 möglichen 2×2-Quadrate ausgegeben werden (wobei in den 4 Positionen die Zahlen von 1 bis 4 stehen). Für $n = 2$ gibt es tatsächlich kein magisches Quadrat.


```java
// ---------------------------------------------
// OOP - Hausaufgabe 7 - Magisches Quadrat
// Name: _____
// ---------------------------------------------

// Fuer Eingaben:
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;

// Fuer Messung der CPU-Zeit:
import java.lang.management.ManagementFactory;
import java.lang.management.ThreadMXBean;

// Klasse zur Erzeugung magischer Quadrate:
class Quadrat {
	// Anzahl gefundener Quadrate:
	private static int numSquares;

	// Maximalzahl zu druckender Quadrate:
	private static final int MAX_PRINT = 20;

	// Pruefung, ob Loesung:
	private static boolean isMagicSquare(int n, int[] square) {
		// Array pruefen:
		if(square == null) {
			System.err.println("Array ist null");
			System.exit(3);
		}
		if(square.length != n*n) {
			System.err.println("Array-Groesse falsch");
			System.exit(3);
		}

		// Pruefen, dass alle Eintraege zwischen 1 und n^2:
		for(int i = 0; i < square.length; i++) {
			if(square[i] <= 0 || square[i] > n*n) {
				System.err.println("Falscher Wert " +
					square[i] + " an Pos. " + i);
				System.exit(3);
			}
		}

		// Richtiges Ergebnis fuer Summen berechnen:
		int s = n * (n*n + 1) / 2;

		// Zeilensummen pruefen:
		for(int i = 0; i < square.length; i = i + n) {
```

```
            if(sum(n, square, i, 1) != s)
                return false;
        }

        // Spaltensummen pruefen:
        for(int i = 0; i < n; i = i + 1) {
            if(sum(n, square, i, n) != s)
                return false;
        }

        // Diagonale von links oben nach rechts unten:
        if(sum(n, square, 0, n+1) != s)
            return false;

        // Diagonale von rechts oben nach links unten:
        if(sum(n, square, n-1, n-1) != s)
            return false;

        // Testen, dass alle Eintraege verschieden:
        for(int i = 0; i < square.length; i++) {
            for(int j = i + 1; j < square.length; j++) {
                if(square[i] == square[j])
                    return false;
            }
        }

        // Alles ok:
        return true;
    }

    // Hilfsfunktion: Summe berechnen
    private static int sum(int n, int[] square,
                int start, int dist) {
        int s = 0;
        int pos = start;
        for(int i = 0; i < n; i++) {
            s = s + square[pos];
            pos = pos + dist;
        }
        return s;
    }

    // Magisches Quadrat drucken:
    private static void printSquare(int n, int[] square) {
        // Anzahl erhoehen:
        numSquares++;

        // Falls Maximalzahl ueberschritten: Fertig.
```



```
        if(numSquares > MAX_PRINT)
            return;

        // Sonst Quadrat drucken:
        for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++)
                System.out.format("%2d␣",
                        square[i*n+j]);
            System.out.println();
        }
        // Leerzeile zur Trennung mehrerer Quadrate:
        System.out.println();
    }

    // Funktion zur Generierung der magischen Quadrate:
    // ...

    // Hilfsmethode zur Eingabe der Kantenlaenge:
    private static String inputLine(String prompt) {
        // Zur Abwechselung mit einem Buffered Reader:
        BufferedReader in = new BufferedReader(
                new InputStreamReader(System.in));
        System.out.print(prompt + ":␣");
        String line = "";
        try {
            line = in.readLine();
        } catch(IOException e) {
            System.err.print("Fehler␣bei␣der␣Eingabe:␣");
            System.err.println(e);
            System.exit(2);
        }
        return line;
    }

    // Hauptprogramm:
    public static void main(String[] args) {

        // Argumente aus Kommandozeile pruefen:
        if(args.length > 1) {
            System.err.println("Zu␣viele␣Argumente␣" +
                "in␣der␣Kommandozeile.");
            System.exit(2);
        }

        // Eingabe beschaffen (Kommandozeile oder Eingabe):
        String input;
        if(args.length == 1)
            input = args[0];
```

```
                else
                        input = inputLine("Bitte␣n␣eingeben");

                // Eingabe in Zahl umwandeln:
                int n = 3;
                try {
                        n = Integer.parseInt(input);
                }
                catch(NumberFormatException e) {
                        System.err.println("Argument␣muss␣" +
                                "eine␣ganze␣Zahl␣sein.");
                        System.exit(2);
                }

                // n muss positiv sein:
                if(n <= 0) {
                        System.err.println("Fehler:␣n␣is␣negativ.");
                        System.exit(2);
                }

                // Beginn der Messung der Realzeit:
                long startTime = System.currentTimeMillis();

                // Magische Quadrate generieren:
                genMagicSquares(n);

                // Verbrauchte CPU-Zeit ausgeben:
                ThreadMXBean tMan =
                        ManagementFactory.getThreadMXBean();
                if(tMan.isCurrentThreadCpuTimeSupported() &&
                                tMan.isThreadCpuTimeEnabled()) {
                        long cpuTime =
                                tMan.getCurrentThreadCpuTime();
                        if(cpuTime != -1) {
                                System.out.println(
                                        "Verbrauchte␣CPU-Zeit:␣" +
                                        (cpuTime / 1000000) + "ms");
                        }
                }

                // Abgelaufene Realzeit ausgeben:
                long realTime = System.currentTimeMillis() -
                                                        startTime;
                if(realTime >= 0) {
                        System.out.println("Abgelaufene␣Realzeit:␣" +
                                        realTime + "ms");
                }

```



```
                // Anzahl gefundener Quadrat ausgeben:
                System.out.println("Anzahl␣gefundener␣Quadrate:␣" +
                                        numSquares);
        }
}
```

Übungsaufgabe 7A: (ohne Abgabe)

Bitte bearbeiten Sie die Übungsaufgaben auf dem Hausaufgabenblatt, aber geben Sie diese Aufgaben nicht ab. Diese Aufgaben werden in der Übung besprochen. Sie müssen Ihre Lösung eventuell in der Übung vorführen.

Das folgende Programm enthält (mindestens) sechs Fehler. Finden Sie möglichst viele davon. Beachten Sie, dass wenn Sie das Programm durch einen Compiler schicken, der Compiler nur einen Teil der Fehler anzeigt. Erst wenn Sie diese korrigieren, werden noch andere angezeigt. Sie sollen aber die Fehler möglichst ohne Compiler finden.

```
class X {
        public static void main(String[] y) {
                double x;
                if(y.length == 0)
                        x = 3;
                double z = f(x);
                if(z + 1)
                        System.out.println("?");
        }

        public static void f(double x) {
                z = g(x);
        }

        public static double g(int x) {
                x = 27;
        }
}
```

Übungsaufgabe 7B: (ohne Abgabe)

Lesen Sie die Dokumentation zum `BufferedReader` und beantworten Sie die Frage, ob er eine Methode zum Lesen eines einzelnen Zeichens hat.

- Wie heißt die Methode?
- Was ist der Ergebnis-Datentyp dieser Methode?
- Hat diese Methode Parameter?
- Könnte die Methode eine Exception auslösen?

Auch diese Aufgabe bitte nicht einsenden, aber bis zur Übung bearbeiten.