Antwort auf Nachfragen 1

Adjazenzmatrix aMatrix = new Adjazenzmatrix(feldGroesse)

hier wird ein neues Objekt vom Typ Adjazenzmatrix erstellt (dadurch dass die Klasse Adjazenzmatrix einen Konstruktor hat, können wir die Klasse wie einen neuen Datentyp behandeln. die Definition muss man aber etwas umständlicher machen.) Alles, was der Konstruktor als eingabe erwartet, kommt in die Klammern.
Da aMatrix jetzt ein Objekt ("Instanz") von der Klasse Adjazenzmatrix ist, kann man über aMatrix.METHODENNAME() die Methoden aus de Klasse aufrufen.

string args

Dieses public static und die string args sind nicht von Bedeutung. Ich weiß nicht im Detail, warum man das braucht, aber der Compiler (übersetzer von Programmcode in Maschienensprache) benutzt das um zu sehen, wo die "Hauptmethode" ist, mit der er anfangen soll...

Varianten

in der Variante eins wird gleich das ganze Problem gelöst (erst linke untere Ecke, dann rechte untere Ecke, dann obere rechte Ecke). Die Variante zwei ist die "ausformulierte" Version von "Wie löse ich das Problem" (weiter unten)
zu Variante 1: erst speichere ich im Array "punkteFolge" die "Eckpunkte" (linke untere Ecke, dann rechte untere Ecke, dann obere rechte Ecke), dann wird nacheinander der Weg von Ecke 1 zu Ecke 2 und danach von Ecke 2 zu Ecke 3 berechnet.
Diese beiden Listen mit den berechneten Wegpunkten werden zusammengeführt und ausgegeben.

Arraylist

Arraylist ist auch ein Datentyp von der Klasse ArrayList. Das tolle an der ArrayList ist, dass man einfach beliebig weitere Einträge anhängen kann und sich nicht (wie beim Array) vorher auf eine Größe festlegen muss. Die Definition einer ArrayList geht so:

```
ArrayList <DATENTYP> NAME = new ArrayList();
```

Wichtigste Funktionen:

```
NAME.get(i); //ruft den Eintrag an Stelle i der Liste auf
NAME.add(j); //fügt einen Eintrag mit Inhalt j hinzu
NAME.remove(i); //löscht den Eintrag an Stelle i
```

Array

ein Array ist defacto eine Tabelle mit bestimmter Länge (und ggf. Höhe, Tiefe, ...)

Definition:

```
DATENTYP NAME [] = new DATENTYP [LÄNGE]
oder
DATENTYP NAME [] = {WERT1, WERT2, WERT3, ...) //die Länge ist dann die Anzahl der angegebenen Werte
oder zweidimensional:
DATENTYP NAME [][] = new DATENTYP [LÄNGE] [HÖHE]

Aufruf (ggf mit zweiter Dimension):
NAME [POSITION]
Verändern (ggf mit zweiter Dimension):
NAME [POSITION] = NEUER WERT
```

Das Lösen von dem Problem geht in zwei Schritten:

- adjazenzmatrix erstellen:

```
Adjazenzmatrix<Adjazenzmatrix> aMatrix = new Adjazenzmatrix(feldGroesse);
        aMatrix.befuellen0();
        aMatrix.befuellenSpringermuster();
```

- Dijkstra berechnen (in der Liste stehen dann in der Reihenfolge die besuchten Felder drinnen):

ArrayList <Integer> Ergebnis = dijkstra(aMatrix, Anfangsfeld, Endfeld);

Bedeutung von this.

this.VARIABLENNAME ruft die Variable aus der übergeordneten Konstruktion (in dem Fall Klasse) auf.
Zum Beispiel die Variable feldGroesse: die wurde einmal in der Klasse erstellt ("gültig"/verwendbar in der ganzen Klasse und untergeordneten Methoden). Zusätzlich gibt es eine andere Variable mit dem Namen feldGroesse auch im Konstruktor (die kann man nur im Konstruktor aufrufen. Dafür kann man im Konstruktor nicht mehr auf die "ursprüngliche" feldGroesse zugreifen, die in der ganzen Klasse gilt, aber im Konstruktor von der eigenen feldGroesse überschrieben wurde.
Damit die anderen Funktionen in der Klasse auch auf diesen, im Konstruktor übergebenen Wert, zugreifen kann, muss die übergeordnete feldGroesse auf den Wert der untergeordneten feldGroesse zugreifen können. Das geht, indem man mit dem this.feldgroesse Befehl auf die übergeordnete feldGroesse zugreift.

befuellenSpringermuster und parse2dto1d

Nummerierung und Schleifen

ich geh die Methode einfach Mal durch.
Die beiden for Schleifen sollen dafür sorgen, dass jedes der Felder auf dem Schachfeld (durch das der Springer springt) einmal besucht wird. Dazu stellen wir uns das als Koordinatensystem vor (Ursprung links oben). Das Schachfeld sieht so aus (mit eingezeichneten Koordinaten und durchnummeriert):

	0	1	2
0	0	1	2
1	3	4	5
2	6	7	8

jedes der nummerierten Felder hat also eine x und y Koordinate (8 z.b. 2|1)
die obere for Schleifen läuft die Zeilen durch. innerhalb jeder Zeile läuft die untere Schleife nochmal durch die Spalten -> jedes Feld kommt der Reihenfolge nach dran.

```
for(int y=0;y<feldGroesse;y++){
    for (int x=0;x<feldGroesse;x++){
```

Idee hinter parse2dto1d:

Weil wir in der Matrix ja auf der x und y Achse jeweils alle Felder haben (von 0 bis 8) müssen wir von der x|y Nummerierung zur 0-8 Nummerierung kommen. Das macht die Methode parse2dto1d indem der y-Wert (Anzahl der schon bearbeiteten Zeilen) mit der feldgroesse multipliziert wird (dann hat man die Feldnummer der aktuellen Zeile) und der x-Wert addiert wird. Beispiel: (1|0) -> 0*3+1 = 1 (1|2) -> 2*3+1 = 7

Exkurs zu Methodendefinition:

Sichtbarkeit	Typ der Rückgabevariable	Name	Eingabevariablen
public static	int	parse2dto1d	int x, int y, int feldGroesse
Definiert Orte im Code, die darauf zugreifen können (ist völlig egal und checkt keiner, machmal gibts deshalb aber komische Fehlermeldungen, deshalb nehme ich immer das, was mir mein Code-Editor vorschlägt)	Methoden können Werte zurückgeben (String, int, boolean,...). Viele Funktionen arbeiten aber nur mit globalen (von überall erreichbare) Variablen und haben dann den Rückgabetyp void (leer)	zu Deutsch: übersetzeZweidimensionaleDarstellungInEindimensionaleDarstellung	selbsterklärend

Exkurs zu (verkürzten) if-else Anweisungen

```
standard if-else Anweisung:
if(x){
y;
}else{
z;
}

verkürzte Form(nur, wenns um Werte geht):
a=  x ? y : z;
```

Umsetzung parse2dto1d

Diese drei Dinge werden in der Funktion kombiniert zu:

```
public static int parse2dTo1d (int x, int y, int feldGroesse){
        return x>=0 && x<feldGroesse && y>=0 && y<feldGroesse ? (x+y*feldGroesse) : -1;
    }
```

Der Teil

```
x>=0 && x<feldGroesse && y>=0 && y<feldGroesse
```

überprüft, ab die angegebenen Koordinaten innerhalb des Feldes liegen (gleich wird klar, warum das nötig ist). Das "return" gibt an, was die Funktion zurückgibt (Rückgabewert).

Verknüpfungen in Matrix setzen und try/catch

```
matrix [parse2dTo1d(x, y)] [parse2dTo1d(x + 1, y + 2)] = 1;
```

Es passiert folgendes: im Array "matrix" wird an der x-Position aktuellesFeld und der y-Position (aktuellesFeld, eins nach rechts, zwei nach unten) der Wert 1 eingetragen. Jetzt ist in der Matrix das aktuelle Feld mit einem Feld, das mit einem Springerzug erreichbar ist, "verbunden". Sollten wir uns allerdings grade im Feld Nummer 8 (2|2) befinden, gibt es das entsprechende Feld "eins nach rechts, zwei nach unten" gar nicht. Dann würde die Funktion parse2dto1d den Wert minus eins ausgeben (da die für sie mitgegebenen x und y Werte außerhalb des Schachfelds liegen). dann versucht das Programm, an die Stelle 8|-1 in der Matrix eine 1 reinzuschreiben. Weil die Matrix aber erst bei 0 anfängt, gibt das Programm einen "ArrayIndexOutOfBoundsException" Fehler aus. Der wird abgefangen und der (von uns herbeigeführte) missglückte Versuch nicht weiter beachtet. Das ganze passiert pro "aktuellem Feld" in jede Richtung, in die der Springer ziehen könnte.

Dijkstra Ablauf

ausgangspunkte	Liste der Indizes (Nummern) der Abzuarbeitenden Punkte
knoten	Array in dem alle Knoten (=Felder auf Schachfeld) gespeichert sind. an Position i is Knoten/das Feld mit der Nummer/Position i
startpunkt	Nummer des Knotens/des Feldes, bei dem der Weg beginnt

- *startpunkt* zu *ausgangspunkte* hinzufügen (Z.51)
- Aus *ausgangspunkte* den Punkt mit kleinstem Wert suchen -> "aktueller Punkt" (Z.65-74)
- Punkte durcharbeiten, die in Adjazenzmatrix mit "1" mit dem aktuellen Punkt verbunden sind ("folgeKnoten") (Z.67-77)
- Vergleiche jeweils, ob bisheriger folgeKnoten.wert im Knotenarray größer als der aktuelle+dem des aktuellen Folgeknotens aus Matrix ist. wenn größer, setze Knotenwert auf aktuellerKnoten.wert+Matrixwert und setze den Vorgänger neu. wenn kleiner, lass den alten wert. (Z.78-80)
- NachfolgerPunkt in *ausgangspunkte* hinzufügen. aktuellen Punkt aus *ausgangspunkte* entfernen. Schritte wiederholen (Z.81-91)
- Abbruch wenn *ausgangspunkte* leer (Z. 64)