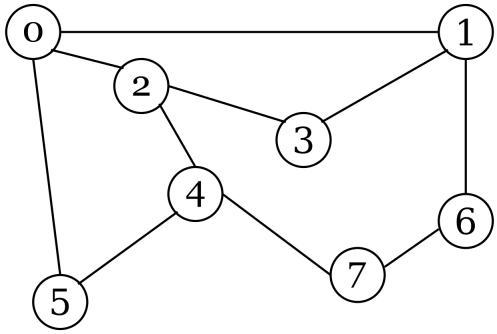
Nach diesen Vorüberlegungen und Grundbegriffen können wir uns der Implementierung eines Graphen widmen. Die Struktur der Knoten bzw. der Speicherung von Daten in den Knoten können wir dabei von Bäumen bzw. Listen übernehmen. Das entscheidende neue Element ist also die Repräsentation der Kanten und deren Gewichte.

Die beiden häufigsten Möglichkeiten der Implementierung verwenden eine **Adjazenzmatrix** bzw. **Adjazenzlisten**, in diesem Kapitel werden wir uns der Matrix widmen.
Wir betrachten wieder ein Beispiel aus den Grundlagen:



Es handelt sich um einen ungerichteten, zusammenhängenden Graphen. Will man die Verbindungen zwischen den einzelnen Knoten darstellen, bietet es sich an „von Knoten zu Knoten“ zu denken. Man betrachtet eine Matrix nach folgendem Schema (die einzelnen Knoten werden zusätzlich zu ihrem Index noch mit k benannt, um später leichtere Lesbarkeit zu gewährleisten):

	k0	k1	k2	k3	k4	k5	k6	k7
k0								
k1								
k2								
k3								
k4								
k5								
k6								
k7								

Wir betrachten exemplarisch die Kante zwischen den Knoten 0 und 2. Häufig interpretiert man die Tabelle so, dass auf der linken Seite - also den Zeilen - die Startknoten abgebildet werden, die Spalten dagegen bilden die Zielknoten. Wollen wir also eintragen, dass eine Kante von 0 nach 2 läuft, suchen wir in der Zeile $k0$ die Spalte $k2$ und markieren die entsprechende Zelle:

	k0	k1	k2	k3	k4	k5	k6	k7
k0			x					
k1								
k2								
k3								
k4								
k5								
k6								
k7								

Da es sich um einen ungerichteten Graphen handelt, geht aber auch ein Weg von 2 nach 0, also muss direkt auch in der Spalte $k0$ und Reihe $k2$ eine Markierung gesetzt werden.
Dies wird für alle Einträge der Fall sein! Man spricht hier auch von einer **symmetrischen** Matrix, d.h. es genügt sich alle Einträge in der „oberen rechten Hälfte“ anzusehen, d.h. von kleineren zu größeren Zahlen. Danach kann „nach unten links“ gespiegelt werden. Zunächst tragen wir aber alle weiteren Kanten von kleinerer zu größerer Zahl ein:

	k0	k1	k2	k3	k4	k5	k6	k7
k0		x	x			x		
k1				x			x	
k2				x	x			
k3								
k4						x		x
k5								
k6								x
k7								

Mit den gespiegelten Werten ergibt sich:

	k0	k1	k2	k3	k4	k5	k6	k7
k0		x	x			x		
k1	x			x			x	
k2	x			x	x			
k3		x	x					
k4			x			x		x
k5	x				x			
k6		x						x
k7					x		x	

Jetzt stellt sich die Frage, wie eine solche Matrix in einem Programm darstellen lässt. Die einfachste Variante ist ein zweidimensionales Array. In Bezug auf den Datentyp des Feldes gibt es verschiedene Möglichkeiten. Für einen ungewichteten Graphen ist nur wichtig, ob eine Verbindung zwischen zwei Knoten vorhanden ist, da die Kante keine zusätzlichen Informationen enthält, man könnte hier also ein Boolean-array verwenden, die Tabelle sähe also wie folgt aus:

	k0	k1	k2	k3	k4	k5	k6	k7
k0	False	True	True	False	False	True	False	False
k1	True	False	False	True	False	False	True	False
k2	True	False	False	True	True	False	False	False
k3	False	True	True	False	False	False	False	False
k4	False	False	True	False	False	True	False	True
k5	True	False	False	False	True	False	False	False
k6	False	True	False	False	False	False	False	True
k7	False	False	False	False	True	False	True	False

Sobald allerdings Kantengewichte eine Rolle spielen, ist ein boolean-array nicht mehr ausreichend, da die Gewichte nicht mehr anderweitig gespeichert werden sollten. Geht man davon aus, dass sich alle Informationen, die man codieren möchte in irgendeiner Form durch Zahlen ausdrücken lassen, so wäre ein **double**[ ][ ]-array geeignet, um möglichst viele Spielarten von Graphentypen abbilden zu können. In unserem Fall ohne Gewichte könnte dann beispielsweise 1 für „Kante existiert“ und 0 für „Kante existiert nicht“ stehen:

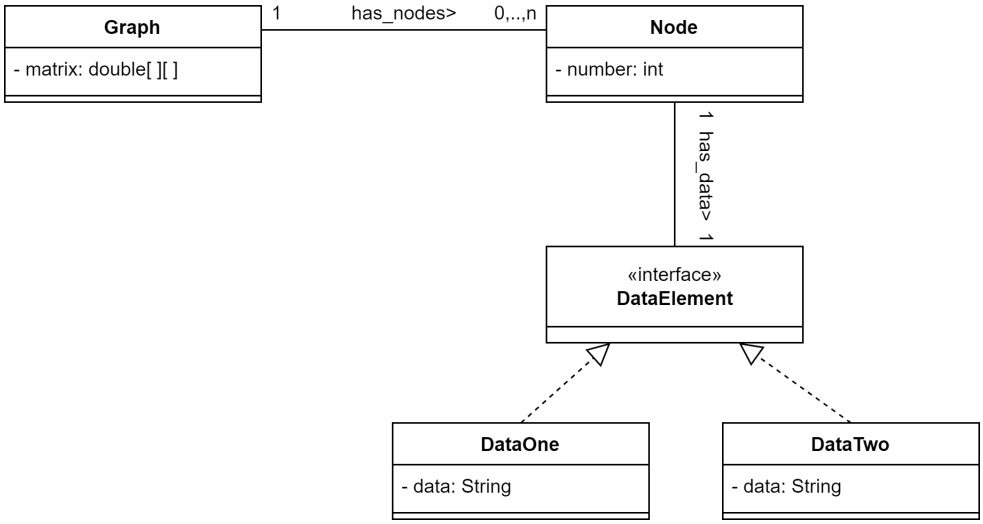
	k0	k1	k2	k3	k4	k5	k6	k7
k0	0	1	1	0	0	1	0	0
k1	1	0	0	1	0	0	1	0
k2	1	0	0	1	1	0	0	0
k3	0	1	1	0	0	0	0	0
k4	0	0	1	0	0	1	0	1
k5	1	0	0	0	1	0	0	0
k6	0	1	0	0	0	0	0	1
k7	0	0	0	0	1	0	1	0

Alternativ könnte auch z.B. -1 und 1 oder jede beliebige andere Kombination verwendet werden.
Diese Implementierung hat natürlich den Nachteil, das bei vielen Knoten, aber wenigen Kanten viele Einträge in der Matrix gleich 0 sind. Man spricht von einer „dünn besetzten Matrix“. Für unsere kleinen Graphen ist das nicht so schlimm, aber möchte man beispielsweise das Straßennetz einer Stadt - oder von ganz Deutschland, oder der

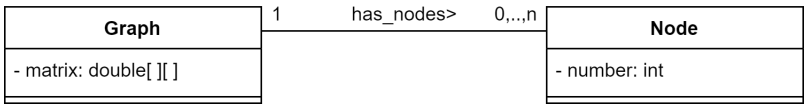
Welt! - modellieren, stellt das einen unnötigen Verbrauch von Speicherplatz dar. In der „Realität“ wird häufig die Idee der Adjazenmatrix beibehalten, aber die Speicherung der Information wird auf andere Datenstrukturen als ein $n \cdot n$ - Feld ausgelagert (mit n der Anzahl der Knoten!). Verwendet werden z.B.:

- **Dictionarys** (DOK: Dictionary of keys: in Java z.B. HashTables, HashMaps): Es werden (Zeile, Spalte)-Paare auf den Kantenwert abgebildet.
- **Coordinate list**: es werden direkt (Zeile, Spalte, Kantenwert)-Tripel gespeichert.
- **Compressed sparse row** (Yale format): Es werden drei Arrays gespeichert: eines für die Kantenwerte, eines für die Spaltenindizes an denen diese stehen und eines für die Zeilenindices.

In allen drei Fällen werden die Null-Werte nicht mehr mitgespeichert, aber zurück zu unserer - deutlich einfacheren - Implementierung. Ein mögliches Klassendiagramm für unseren Graphen sieht wie folgt aus:



Das Attribut **maxNodes** ist nicht zwingend notwendig, es hängt davon ab, ob man den Graphen erweiterbar gestalten möchte oder nicht. Der Vorteil der Verwendung eines Maximums besteht darin, dass das zweidimensionale Feld direkt mit seiner endgültigen Größe erzeugt werden kann. Ansonsten müsste beim Hinzufügen eines Knotens immer die Adjazenzmatrix vergrößert werden, was nur durch Erzeugen eines neuen Arrays und Kopieren des Alten Arrays möglich wäre.
Im Folgenden wollen wir uns außerdem darauf beschränken, dass die Knoten des Graphen keine weiteren Informationen verwalten müssen außer ihrer eigenen Nummer. Wir wollen außerdem bei Erstellung des Graphen festlegen, wie groß er ist, da dies der Regelfall im Einsatz ist, d.h. ein grundlegendes Klassendiagramm unseres einfachstmöglichen Graphen sieht (noch ohne Methoden) wie folgt aus:



Aufgabe 1: Implementieren Sie die Grundstruktur des Graphen mit Adjazenzmatrix nach obigem vereinfachten Klassendiagramm, zusätzlich soll es folgendes geben:

- einen Konstruktor, der einen Graphen mit einer bestimmten Anzahl an Knoten erzeugt.
- einen Konstrukter, der einen Graphen erzeugt, wenn eine Knotenliste und eine Adjazenzmatrix übergeben werden.
- eine Methode, die eine Kante ohne Gewicht hinzufügt.
- eine Methode, die eine Kante mit Gewicht hinzufügt.

- für die beiden vorangehenden Methoden jeweils eine Methode, die außerdem die „Rückrichtung“ gleich mit setzt (für ungerichtete Graphen).
- eine Methode, die einen Knoten hinzufügt, wenn die Maximalzahl noch nicht erreicht ist.
- eine Methode, die prüft, ob zwischen zwei Knoten eine Kante existiert.
- eine Methode, die eine existierende Kante löscht und eine Option, die eine Kante in beide Richtungen löscht!.
- eine Methode, die die Adjazenzmatrix lesbar auf der Konsole ausgibt.

Die Lösung zu dieser Aufgabe findet sich auf der nächsten Seite.

Wir beginnen mit der Knoten-Klasse, hier gibt es aktuell noch nicht viel zu tun:

```
public class Node {

    private int number;

    public Node(int number) {
        this.number = number;
    }
}
```

In der Graphen-Klasse sind die Konstruktoren zuerst an der Reihe:

```
public class Graph {

    private double[][] matrix;
    private Node[] nodes;

    public Graph(double[][] matrix, Node[] nodes) {
        this.nodes = nodes;
        this.matrix = matrix;
    }


    public Graph(int nodeNumber) {
        if(nodeNumber <= 0) nodeNumber = 5;
        nodes = new Node[nodeNumber];
        for(int i = 0; i < nodeNumber; i++) {
            nodes[i] = new Node(i);
        }
        matrix = new double[nodeNumber][nodeNumber];
    }
}
```

Im zweiten Fall werden die Knoten von 0 bis $nodeNumber - 1$ erzeugt und im Knoten-Array gespeichert. Wir können ein Array verwenden, da wir davon ausgehen, dass sich die Größe unseres Graphen nicht mehr verändert. Ansonsten könnten wir natürlich auch eine Liste verwenden!

Die Methoden um die Kanten hinzuzufügen sind vergleichsweise simpel: es muss jeweils in der Matrix an der korrekten Stelle entweder eine 1 platziert werden, oder der Wert des übergebenen Gewichts für diese Kante. Damit ergeben sich die folgenden vier Methoden (jeweils noch eine zusätzlich, um gleich beide Kanten für einen ungerichteten Graphen zu erzeugen). Die vorangestellte checkInput()-Methode wird dazu verwendet, in jedem Fall zu prüfen, ob die übergebenen Indices zu unserem Knotenarray passen:

```
public boolean checkInput(int start, int end) {
    if(start < 0 || end < 0 || start > nodes.length || end > nodes.length) {
        System.out.println("There cannot be an edge here!");
        return false;
    }
    return true;
}

public void addEdge(int start, int end) {
    if(checkInput(start, end)) matrix[start][end] = 1;
}

public void addEdge(int start, int end, double weight) {
```

```
    if(checkInput(start, end)) matrix[start][end] = weight;
}

public void addEdgeBoth(int start, int end) {
    if(checkInput(start, end)) {
    matrix[start][end] = 1;
    matrix[end][start] = 1;
    }
}

public void addEdgeBoth(int start, int end, double weight) {
    if(checkInput(start, end)) {
        matrix[start][end] = weight;
        matrix[end][start] = weight;
    }
}
```

Wir gehen hier davon aus, dass die einzige relevante Information eines Knotens seine Nummer ist, mit der wir ihn erzeugt haben. Da diese Nummer auch der Position im Array entspricht (deswegen der Beginn bei 0), können wir den Methoden hier direkt die Positionen im Array übergeben, um die Kanten zu definieren. Möchte man sich nicht auf diese Reihenfolge verlassen bzw. sie benutzen, so muss jeder Knoten mit einem anderen Attribut eindeutig zuordbar sein. Dann kann jeweils dieses Attribut der *addEdge()* - Methode übergeben werden und die entsprechenden Knoten können im Array gesucht werden.
Bei der Methode, die prüft, ob eine Kante existiert, gibt es nicht viel zu beachten, ebensowenig bei den Methoden zum Löschen:

```
public boolean existsEdge(int start, int end) {
    if(!checkInput(start, end)) return false;
    if(matrix[start][end] != 0) {
        return true;
    }
    return false;
}

public void removeEdge(int start, int end) {
    if(checkInput(start, end)) matrix[start][end] = 0;
}

public void removeEdgeBoth(int start, int end) {
    if(checkInput(start, end)) {
        matrix[start][end] = 0;
        matrix[end][start] = 0;
    }
}
```

Die Ausgabe-Methode schreibt zusätzlich zu den Gewichten noch die Knotennamen in die oberste Spalte bzw. Zeile. Der Tabstop wird verwendet, damit die Zahlen hübscher untereinander angezeigt werden.

```
public void printMatrix() {
    System.out.print("\t");
    for(int i = 0; i < nodes.length; i++) {
        System.out.print("k" + i + "\t");
    }
    for(int i = 0; i < nodes.length; i++) {
        System.out.println();
        System.out.print("k" + i + "\t");
        for(int j = 0; j < nodes.length; j++) {
```

```
            System.out.print(matrix[i][j] + "\t");
        }
    }
}
```

Alle weiteren möglicherweise interessanten Methoden, werden wir in Kapitel 4 - Graphenalgorithmen - implementieren.