Einführung in die Programmierung

Vorlesungsteil 7

Felder und Graphen

Umfangreiche Datenbestände erfassen und verarbeiten

PD Dr. Thomas Hinze

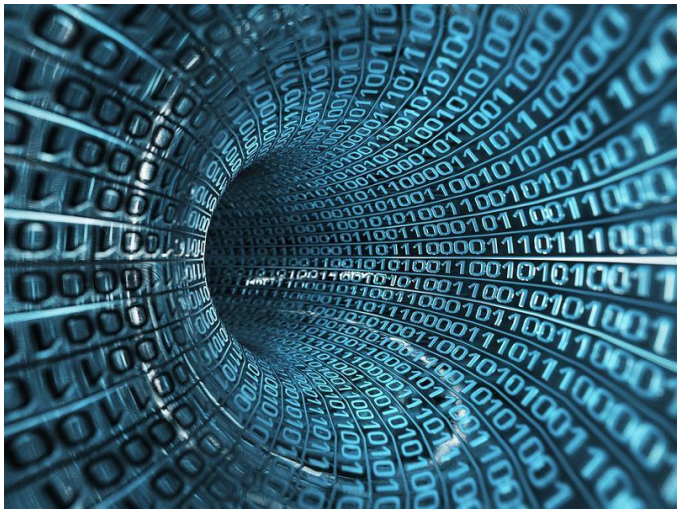
Brandenburgische Technische Universität Cottbus – Senftenberg
Institut für Informatik

Sommersemester 2016

Brandenburgische
Technische Universität
Cottbus - Senftenberg

Große Datenmengen sind häufig zu verarbeiten

www.bund.de



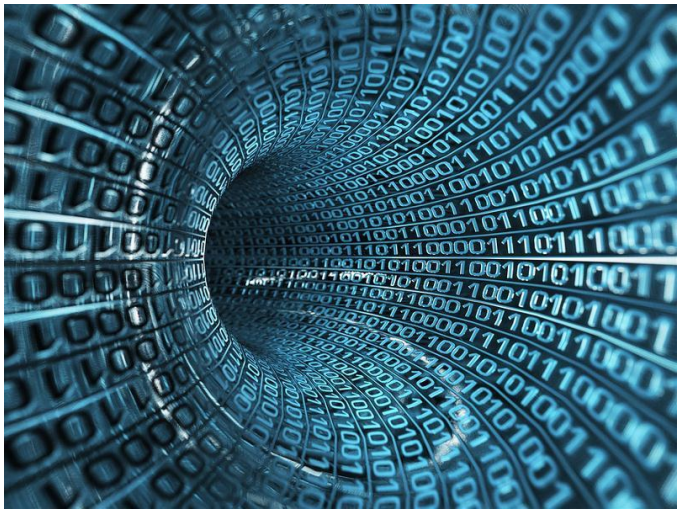
www.ingenieur.de

- Weltweites Datenvolumen verdoppelt sich etwa alle zwei Jahre*

* : Studie der UC Berkeley School of Information, 2013

Große Datenmengen sind häufig zu verarbeiten

www.bund.de



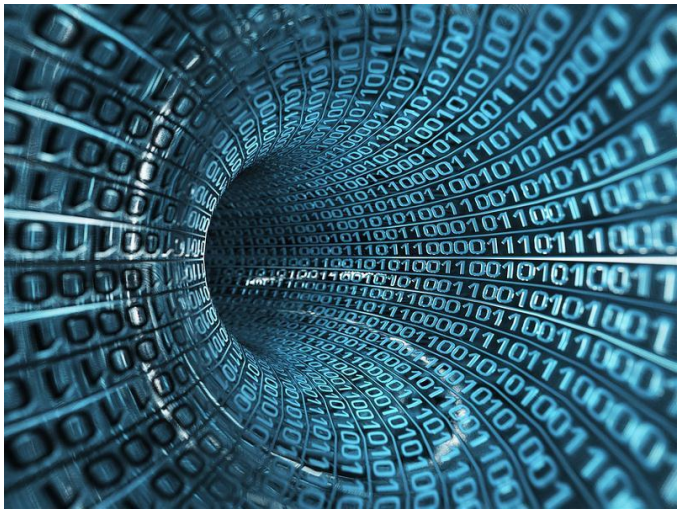
www.ingenieur.de

- Weltweites Datenvolumen verdoppelt sich etwa alle zwei Jahre*
- Weltweiter Datenbestand auf gegenwärtig etwa $5 \cdot 10^{21}$ Bytes (5 Zettabytes) geschätzt*

*: Studie der UC Berkeley School of Information, 2013

Große Datenmengen sind häufig zu verarbeiten

www.bund.de



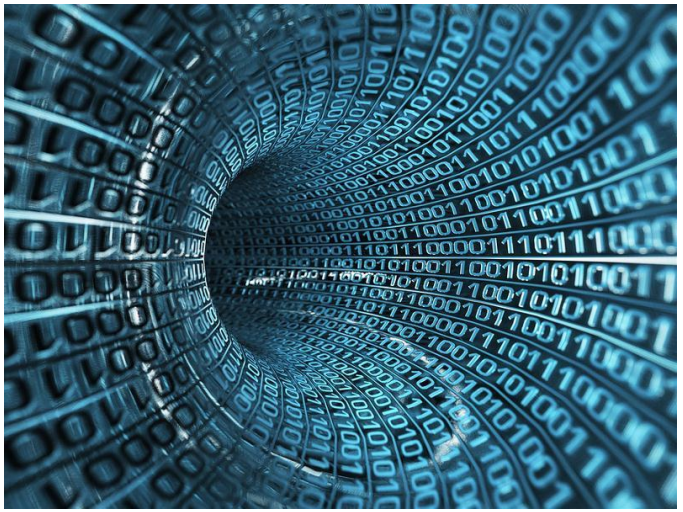
www.ingenieur.de

- Weltweites Datenvolumen verdoppelt sich etwa alle zwei Jahre*
- Weltweiter Datenbestand auf gegenwärtig etwa $5 \cdot 10^{21}$ Bytes (5 Zettabytes) geschätzt*
- Speicherkapazität des gesunden menschlichen Gehirns bei etwa 10^{15} Bytes (15 Petabytes) vermutet

* : Studie der UC Berkeley School of Information, 2013

Große Datenmengen sind häufig zu verarbeiten

www.bund.de



www.ingenieur.de

- Weltweites Datenvolumen verdoppelt sich etwa alle zwei Jahre*
- Weltweiter Datenbestand auf gegenwärtig etwa $5 \cdot 10^{21}$ Bytes (5 Zettabytes) geschätzt*
- Speicherkapazität des gesunden menschlichen Gehirns bei etwa 10^{15} Bytes (15 Petabytes) vermutet
- *Big Data* als vielversprechendes Forschungsgebiet

*: Studie der UC Berkeley School of Information, 2013

Umfangreiche Datenmengen effizient verarbeiten

Angenommen, Sie haben 12 Messwerte

```
4.31, 4.72, 4.49, 4.18, 4.07, 4.13,
4.56, 4.38, 4.71, 4.52, 4.64, 4.45
```

und wollen daraus den Durchschnitt (arithmetisches Mittel) berechnen.

Umfangreiche Datenmengen effizient verarbeiten

Angenommen, Sie haben 12 Messwerte

```
4.31, 4.72, 4.49, 4.18, 4.07, 4.13,
4.56, 4.38, 4.71, 4.52, 4.64, 4.45
```

und wollen daraus den Durchschnitt (arithmetisches Mittel) berechnen.

- Allein durch Nutzung elementarer Datentypen müssten wir dafür 12 *verschiedene Variablen* anlegen.

Umfangreiche Datenmengen effizient verarbeiten

Angenommen, Sie haben 12 Messwerte

```
4.31, 4.72, 4.49, 4.18, 4.07, 4.13,
4.56, 4.38, 4.71, 4.52, 4.64, 4.45
```

und wollen daraus den Durchschnitt (arithmetisches Mittel) berechnen.

- Allein durch Nutzung elementarer Datentypen müssten wir dafür 12 *verschiedene Variablen* anlegen.
- Die Implementierung der Formel zur Berechnung des Durchschnitts wäre *umständlich* hinzuschreiben.

Umfangreiche Datenmengen effizient verarbeiten

Angenommen, Sie haben 12 Messwerte

```
4.31, 4.72, 4.49, 4.18, 4.07, 4.13,
4.56, 4.38, 4.71, 4.52, 4.64, 4.45
```

und wollen daraus den Durchschnitt (arithmetisches Mittel) berechnen.

- Allein durch Nutzung elementarer Datentypen müssten wir dafür 12 *verschiedene Variablen* anlegen.
- Die Implementierung der Formel zur Berechnung des Durchschnitts wäre *umständlich* hinzuschreiben.
- Angenommen, es gäbe hin und wieder nur 11 oder vielleicht auch einmal 13 Messwerte. Wir müssten dann jedesmal *umfangreiche Änderungen im Quelltext* vornehmen, um auf diese Situationen reagieren zu können.

Feld

Ein *Feld* (engl. *array*) gibt uns die Möglichkeit, eine beliebig große, aber bekannte Anzahl von Datenwerten über einen einheitlichen Bezeichner zu erfassen und auf jeden einzelnen dieser Datenwerte direkt lesend oder schreibend zuzugreifen.

Vorlesung Einführung in die Programmierung mit Java

1. **Einführung und erste Schritte** . . . . . Installation Java-Compiler, ein erstes Programm: HalloWelt, Blick in den Computer
2. **Elementare Datentypen, Variablen, Arithmetik, Typecast** . . . . . Java als Taschenrechner nutzen, Tastatureingabe → Formelberechnung → Ausgabe
3. **Imperative Kontrollstrukturen** . . . . . Befehlsfolgen, Verzweigungen, Schleifen und logische Ausdrücke programmieren
4. **Methoden selbst programmieren** . . . . . Methoden als wiederverwendbare Funktionen, Werteübernahme und -rückgabe
5. **Rekursion** . . . . . selbstaufrufende Funktionen als elegantes algorithmisches Beschreibungsmittel
6. **Objektorientiert programmieren** . . . . . Klassen, Objekte, Attribute, Methoden, Sichtbarkeit, Vererbung, Polymorphie
7. **Felder und Graphen** . . . . . effizientes Handling größerer Datenmengen und Beschreibung von Netzwerken
8. **Sortieren** . . . . . klassische Sortierverfahren im Überblick, Laufzeit und Speicherplatzbedarf
9. **Zeichenketten, Dateiarbeit, Ausnahmen** . . . . . Texte analysieren, ver-/entschlüsseln, Dateien lesen/schreiben, Fehler behandeln
10. **Dynamische Datenstruktur „Lineare Liste“** . . . . . unsere selbstprogrammierte kleine Datenbank
11. **Ausblick und weiterführende Konzepte** . . . . .

Eindimensionales Feld anlegen und initialisieren

(Messwertfeld.java)

```
public class Messwertfeld {
  public static void main(String[] args) {
    double[] messwert = { 4.31, 4.72, 4.49, 4.18, 4.07, 4.13,
                          4.56, 4.38, 4.71, 4.52, 4.64, 4.45 };

    int i;

    for (i = 0; i < messwert.length; i++)
    {
      System.out.printf("%2d   %.2f\n", i, messwert[i]);
    } // for
  } // main
} // class
```

```
 0   4.31
 1   4.72
 2   4.49
 3   4.18
 4   4.07
 5   4.13
 6   4.56
 7   4.38
 8   4.71
 9   4.52
10   4.64
11   4.45
```

- `double[] messwert = {...}` legt Feld an und trägt Messwerte fortlaufend als Feldelemente ein
- `messwert.length` liefert Anzahl Feldelemente (hier: 12 mit den Indizes $0, \ldots, 11$)
- `messwert[i]` liefert Feldelement mit Index `i`

Arithmetisches Mittel aus den Messwerten berechnen

(MesswertfeldDurchschnitt.java)

```
public class MesswertfeldDurchschnitt {
  public static void main(String[] args) {
    double[] messwert = { 4.31, 4.72, 4.49, 4.18, 4.07, 4.13,
                          4.56, 4.38, 4.71, 4.52, 4.64, 4.45 };

    int i;
    double summe = 0.0;

    for (i = 0; i < messwert.length; i++)
    {
      summe = summe + messwert[i];
    }
    System.out.printf("Durchschnitt der Messwerte: %f\n", summe / messwert.length);
  } // main
} // class
```

```
Durchschnitt der Messwerte: 4.430000
```

- Feld elementweise *durchlaufen* und Einträge *aufsummieren*
- Summe anschließend durch Anzahl Feldelemente *teilen*

Maximalen Messwert bestimmen

(MesswertfeldMaximum.java)

```
public class MesswertfeldMaximum {
  public static void main(String[] args) {
    double[] messwert = { 4.31, 4.72, 4.49, 4.18, 4.07, 4.13,
                          4.56, 4.38, 4.71, 4.52, 4.64, 4.45 };

    int i;
    double max = messwert[0];

    for (i = 1; i < messwert.length; i++)
    {
      if (max < messwert[i])
      {
        max = messwert[i];
      }
    }
    System.out.printf("Groesster Messwert: %.2f\n", max);
  } // main
} // class
```

```
Groesster Messwert: 4.72
```

Begriff Feld in der Programmierung

Ein **Feld** (engl. array) bezeichnet in der Programmierung eine *Zusammenfassung von Speicherplätzen* (Variablenwerten) *gleichen Typs*, die über einen oder mehrere *Indizes* angesprochen werden können.

Begriff Feld in der Programmierung

Ein **Feld** (engl. array) bezeichnet in der Programmierung eine *Zusammenfassung von Speicherplätzen* (Variablenwerten) *gleichen Typs*, die über einen oder mehrere *Indizes* angesprochen werden können.

- Die lückenlose Nummerierung der Feldelemente heißt *Index* (Plural: Indizes) und beginnt immer bei 0.

Begriff Feld in der Programmierung

Ein **Feld** (engl. array) bezeichnet in der Programmierung eine *Zusammenfassung von Speicherplätzen* (Variablenwerten) *gleichen Typs*, die über einen oder mehrere *Indizes* angesprochen werden können.

- Die lückenlose Nummerierung der Feldelemente heißt *Index* (Plural: Indizes) und beginnt immer bei 0.
- Der Index ist immer vom Ganzzahltyp und wird in eckige Klammern `[...]` geschrieben.

Begriff Feld in der Programmierung

Ein **Feld** (engl. array) bezeichnet in der Programmierung eine *Zusammenfassung von Speicherplätzen* (Variablenwerten) *gleichen Typs*, die über einen oder mehrere *Indizes* angesprochen werden können.

- Die lückenlose Nummerierung der Feldelemente heißt *Index* (Plural: Indizes) und beginnt immer bei 0.
- Der Index ist immer vom Ganzzahltyp und wird in eckige Klammern `[...]` geschrieben.
- Ein Feld lässt sich durch fortlaufend durchnummerierte Schubfächer veranschaulichen.

Begriff Feld in der Programmierung

Ein **Feld** (engl. array) bezeichnet in der Programmierung eine *Zusammenfassung von Speicherplätzen* (Variablenwerten) *gleichen Typs*, die über einen oder mehrere *Indizes* angesprochen werden können.

- Die lückenlose Nummerierung der Feldelemente heißt *Index* (Plural: Indizes) und beginnt immer bei 0.
- Der Index ist immer vom Ganzzahltyp und wird in eckige Klammern `[...]` geschrieben.
- Ein Feld lässt sich durch fortlaufend durchnummerierte Schubfächer veranschaulichen.
- Mathematische Vorbilder für Felder sind *Vektoren*, *endliche Zahlenfolgen* und *Matrizen*.

Begriff Feld in der Programmierung

Ein **Feld** (engl. array) bezeichnet in der Programmierung eine *Zusammenfassung von Speicherplätzen* (Variablenwerten) *gleichen Typs*, die über einen oder mehrere *Indizes* angesprochen werden können.

- Die lückenlose Nummerierung der Feldelemente heißt *Index* (Plural: Indizes) und beginnt immer bei 0.
- Der Index ist immer vom Ganzzahltyp und wird in eckige Klammern `[...]` geschrieben.
- Ein Feld lässt sich durch fortlaufend durchnummerierte Schubfächer veranschaulichen.
- Mathematische Vorbilder für Felder sind *Vektoren*, *endliche Zahlenfolgen* und *Matrizen*.
- Die Anzahl Feldelemente braucht in Java erst zur Laufzeit bekannt zu sein.

Begriff Feld in der Programmierung

Ein **Feld** (engl. array) bezeichnet in der Programmierung eine *Zusammenfassung von Speicherplätzen* (Variablenwerten) *gleichen Typs*, die über einen oder mehrere *Indizes* angesprochen werden können.

- Die lückenlose Nummerierung der Feldelemente heißt *Index* (Plural: Indizes) und beginnt immer bei 0.
- Der Index ist immer vom Ganzzahltyp und wird in eckige Klammern `[...]` geschrieben.
- Ein Feld lässt sich durch fortlaufend durchnummerierte Schubfächer veranschaulichen.
- Mathematische Vorbilder für Felder sind *Vektoren*, *endliche Zahlenfolgen* und *Matrizen*.
- Die Anzahl Feldelemente braucht in Java erst zur Laufzeit bekannt zu sein.
- Die Anzahl Feldelemente ist nachträglich nicht mehr änderbar.

Felddurchlaufungen vereinfacht notieren mit foreach

Betrachten wir erneut das Programm **Messwertfeld.java**:

```
public class Messwertfeld {
  public static void main(String[] args) {
    double[] messwert = {4.31, 4.72, 4.49, 4.18, 4.07, 4.13,
                         4.56, 4.38, 4.71, 4.52, 4.64, 4.45};

    int i;

    for (i = 0; i < messwert.length; i++)
    {
      System.out.printf("%.2f\n", messwert[i]);
    } // for
  } // main
} // class
```

Felddurchlaufungen vereinfacht notieren mit foreach

Betrachten wir erneut das Programm **`Messwertfeld.java`**:

```
public class Messwertfeld {
  public static void main(String[] args) {
    double[] messwert = {4.31, 4.72, 4.49, 4.18, 4.07, 4.13,
                         4.56, 4.38, 4.71, 4.52, 4.64, 4.45};

    //int i;

    for (double e : messwert)
    {
      System.out.printf("%.2f\n", e);
    } // for
  } // main
} // class
```

Variable `e` läuft („iteriert“) elementweise durch das gesamte Feld, ohne dass ein Index angegeben werden muss. Index-Variable `i` nicht mehr benötigt.

Feldelemente per Direktzugriff mit Werten belegen

In Java ist jedes Feld ein *Objekt* einer (vordefinierten) Array-Klasse

```
public class Quadratzahlen {
  public static void main(String[] args) {

    long[] quadratzahlen = new long[11]; //Speicherplatz fuer Feld ausfassen
                                         //11 Feldelemente vorsehen (0...10)
                                         //Feld quadratzahlen ist ein Objekt
    int i;

    for (i = 0; i < quadratzahlen.length; i++)
    {
      quadratzahlen[i] = i * i;  //Feldelemente mit Werten belegen
    }
    for (i = 0; i < quadratzahlen.length; i++)
    {
      System.out.printf("%2d  %3d\n", i, quadratzahlen[i]);
    }
  } // main
} // class
```

```
 0    0
 1    1
 2    4
 3    9
 4   16
 5   25
 6   36
 7   49
 8   64
 9   81
10  100
```

`long[] quadratzahlen = new long[11]` legt Feld als Objekt an mit gewünschter Feldgröße (hier: 11 Elemente) und gewünschtem Typ der Elemente (hier: **`long`**). Alle vordefinierten Methoden wie z.B. **`length`** stehen dann zur Verfügung und sind auf das Objekt anwendbar.

Feld an eine Methode übergeben

```
public class Quadratzahlen2 {

  public static void ausgabe(long[] feld) //operiert im Speicherbereich
  {                                       //des Feldes quadratzahlen
    int k;

    for (k = 0; k < feld.length; k++)
    {
      System.out.printf("%2d  %3d\n", k, feld[k]);
    }
  }

  public static void main(String[] args)
  {
    long[] quadratzahlen = new long[11]; //Feld anlegen
    int i;

    for (i = 0; i < quadratzahlen.length; i++)
    {
      quadratzahlen[i] = i * i;  //Feldelemente mit Werten belegen
    }
    ausgabe(quadratzahlen); //Anfangsadresse des Feldes wird uebergeben
  } // main
} // class
```

Lediglich *Anfangsadresse* des Feldes übergeben.
Feldelemente werden **nicht** kopiert.

Feld auch als Rückgabe einer Methode möglich

```
public class Quadratzahlen3 {

  public static long[] feldfueller()
  {
    long[] quadratzahlen = new long[11]; //Feld anlegen
    int i;

    for (i = 0; i < quadratzahlen.length; i++)
    {
      quadratzahlen[i] = i * i;  //Feldelemente mit Werten belegen
    }
    return quadratzahlen; //Anfangsadresse des befuellten Feldes zurueckgeben
  }

  public static void main(String[] args)
  {
    long[] zahlenfeld = feldfueller(); //befuelltes Feld entgegennehmen
    for(long e : zahlenfeld)           //Feldinhalt anzeigen
    {
      System.out.println(e);
    }
  } // main
} // class
```

- Feld **`quadratzahlen`** angelegt und belegt in Methode **`feldfueller`**
- Anfangsadresse dieses Feldes per **`return`** zurückgegeben und in **`main`**-Methode entgegengenommen
- Feld jetzt in **`main`**-Methode unter dem Namen **`zahlenfeld`** uneingeschränkt nutzbar.

Zweidimensionales Feld anlegen, befüllen, übergeben

(Kleines1x1.java) – Kleines Einmaleins, Zeilen i: 0 . . . 10, Spalten k: 0 . . . 15

```
public class Kleines1x1 {

  public static void main(String[] args) {
    long[][] produkte = new long[11][16];
    int zeilen = produkte.length;     // 11 Zeilen
    int spalten = produkte[0].length; // 16 Spalten

    for (int i = 0; i < zeilen; i++)
    {
      for (int k = 0; k < spalten; k++)
      {
        produkte[i][k] = i * k;
      }
    }
    ausgabe(produkte);
  } // main

  public static void ausgabe(long[][] zahlentabelle) {
    for (int i = 0; i < zahlentabelle.length; i++)
    {
      for (int k = 0; k < zahlentabelle[0].length; k++)
      {
        System.out.printf("%3d  ", zahlentabelle[i][k]);
      }
      System.out.println();
    }
  } // ausgabe
} // class
```

```
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
  0   2   4   6   8  10  12  14  16  18  20  22  24  26  28  30
  0   3   6   9  12  15  18  21  24  27  30  33  36  39  42  45
  0   4   8  12  16  20  24  28  32  36  40  44  48  52  56  60
  0   5  10  15  20  25  30  35  40  45  50  55  60  65  70  75
  0   6  12  18  24  30  36  42  48  54  60  66  72  78  84  90
  0   7  14  21  28  35  42  49  56  63  70  77  84  91  98 105
  0   8  16  24  32  40  48  56  64  72  80  88  96 104 112 120
  0   9  18  27  36  45  54  63  72  81  90  99 108 117 126 135
  0  10  20  30  40  50  60  70  80  90 100 110 120 130 140 150
```

- Zweidimensionales Feld **`produkte`** verkörpert Matrix oder Tabelle aus Zeilen und Spalten
- Zweidimensionales Feld entspricht einem eindimensionalen Feld von eindimensionalen Feldern.

Eigenschaften jedes Feldes

- Alle Feldelemente besitzen den *gleichen Datentyp*.

Eigenschaften jedes Feldes

- Alle Feldelemente besitzen den *gleichen Datentyp*.
- Die *Größe* des Feldes (Anzahl Elemente und Dimensionierung) muss in Java erst zur *Laufzeit des Programms bekannt sein*.

Eigenschaften jedes Feldes

- Alle Feldelemente besitzen den *gleichen Datentyp*.
- Die *Größe* des Feldes (Anzahl Elemente und Dimensionierung) muss in Java erst zur *Laufzeit des Programms bekannt sein*.
- Jeder *Feldindex* beginnt bei 0.

Eigenschaften jedes Feldes

- Alle Feldelemente besitzen den *gleichen Datentyp*.
- Die *Größe* des Feldes (Anzahl Elemente und Dimensionierung) muss in Java erst zur *Laufzeit des Programms bekannt sein*.
- Jeder *Feldindex* beginnt bei 0.
- Über den Feldindex besteht direkter *wahlfreier Zugriff* auf jedes Feldelement.

Eigenschaften jedes Feldes

- Alle Feldelemente besitzen den *gleichen Datentyp*.
- Die *Größe* des Feldes (Anzahl Elemente und Dimensionierung) muss in Java erst zur *Laufzeit des Programms bekannt sein*.
- Jeder *Feldindex* beginnt bei 0.
- Über den Feldindex besteht direkter *wahlfreier Zugriff* auf jedes Feldelement.
- Der Feldindex ist immer von einem Ganzzahltyp und wird stets in eckige Klammern `[...]` geschrieben.

Eigenschaften jedes Feldes

- Alle Feldelemente besitzen den *gleichen Datentyp*.
- Die *Größe* des Feldes (Anzahl Elemente und Dimensionierung) muss in Java erst zur *Laufzeit des Programms bekannt sein*.
- Jeder *Feldindex* beginnt bei 0.
- Über den Feldindex besteht direkter *wahlfreier Zugriff* auf jedes Feldelement.
- Der Feldindex ist immer von einem Ganzzahltyp und wird stets in eckige Klammern `[...]` geschrieben.
- Ein Feld darf beliebig, aber endlich viele *Dimensionen* haben (mehr als vier Dimensionen aber sehr selten).

Eigenschaften jedes Feldes

- Alle Feldelemente besitzen den *gleichen Datentyp*.
- Die *Größe* des Feldes (Anzahl Elemente und Dimensionierung) muss in Java erst zur *Laufzeit des Programms bekannt sein*.
- Jeder *Feldindex* beginnt bei 0.
- Über den Feldindex besteht direkter *wahlfreier Zugriff* auf jedes Feldelement.
- Der Feldindex ist immer von einem Ganzzahltyp und wird stets in eckige Klammern `[...]` geschrieben.
- Ein Feld darf beliebig, aber endlich viele *Dimensionen* haben (mehr als vier Dimensionen aber sehr selten).
- Die Elemente eines Feldes sind im Speicher *unmittelbar aufeinanderfolgend* abgelegt.

Eigenschaften jedes Feldes

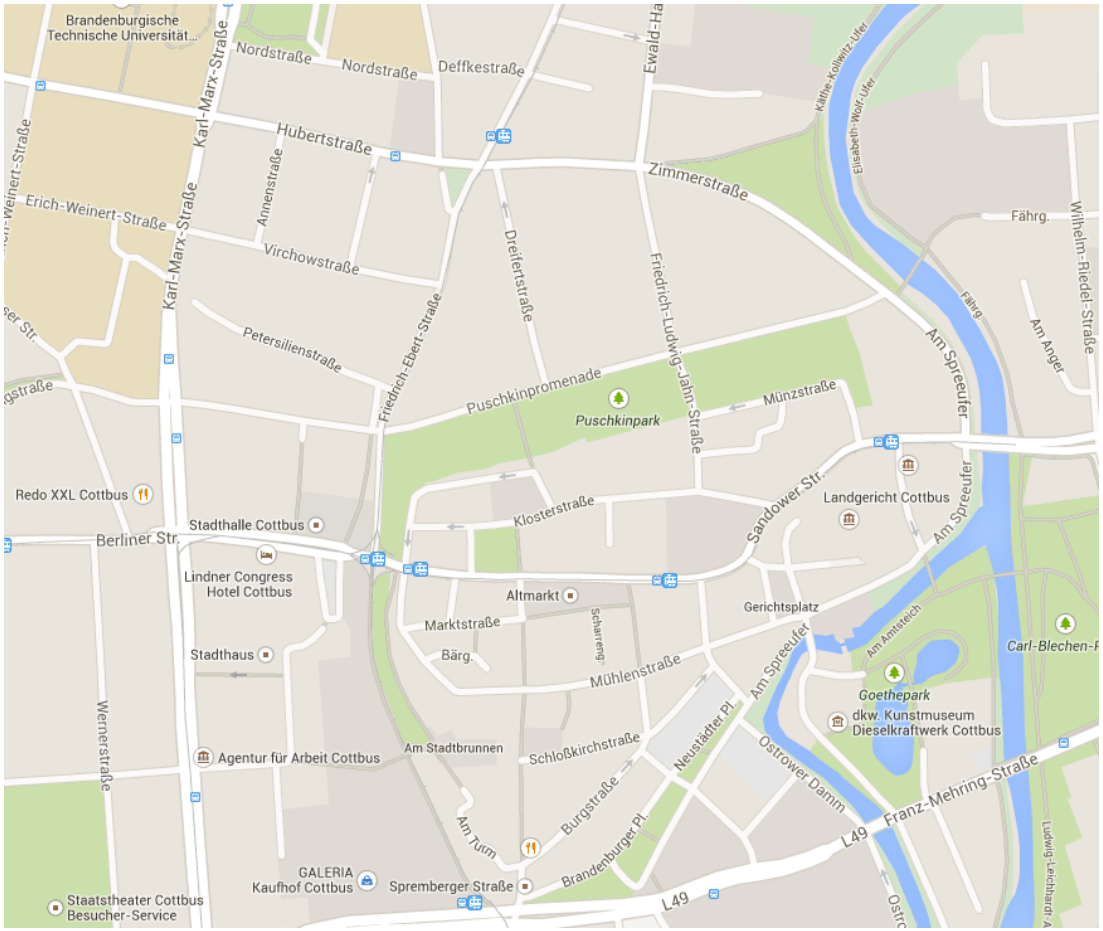
- Alle Feldelemente besitzen den *gleichen Datentyp*.
- Die *Größe* des Feldes (Anzahl Elemente und Dimensionierung) muss in Java erst zur *Laufzeit des Programms bekannt sein*.
- Jeder *Feldindex* beginnt bei 0.
- Über den Feldindex besteht direkter *wahlfreier Zugriff* auf jedes Feldelement.
- Der Feldindex ist immer von einem Ganzzahltyp und wird stets in eckige Klammern `[...]` geschrieben.
- Ein Feld darf beliebig, aber endlich viele *Dimensionen* haben (mehr als vier Dimensionen aber sehr selten).
- Die Elemente eines Feldes sind im Speicher *unmittelbar aufeinanderfolgend* abgelegt.
- Der gewählte *Feldname* verkörpert in Java die *Anfangsadresse*, ab der das Feld abgespeichert ist.

Eigenschaften jedes Feldes

- Alle Feldelemente besitzen den *gleichen Datentyp*.
- Die *Größe* des Feldes (Anzahl Elemente und Dimensionierung) muss in Java erst zur *Laufzeit des Programms bekannt sein*.
- Jeder *Feldindex* beginnt bei 0.
- Über den Feldindex besteht direkter *wahlfreier Zugriff* auf jedes Feldelement.
- Der Feldindex ist immer von einem Ganzzahltyp und wird stets in eckige Klammern `[...]` geschrieben.
- Ein Feld darf beliebig, aber endlich viele *Dimensionen* haben (mehr als vier Dimensionen aber sehr selten).
- Die Elemente eines Feldes sind im Speicher *unmittelbar aufeinanderfolgend* abgelegt.
- Der gewählte *Feldname* verkörpert in Java die *Anfangsadresse*, ab der das Feld abgespeichert ist.
- Bei *Parameterübergaben* wird ein Feld nicht elementweise kopiert, sondern lediglich seine *Anfangsadresse weitergegeben*.

Netzwerk-Strukturen im Computer erfassen

Beispiel: Stadtplan Cottbus

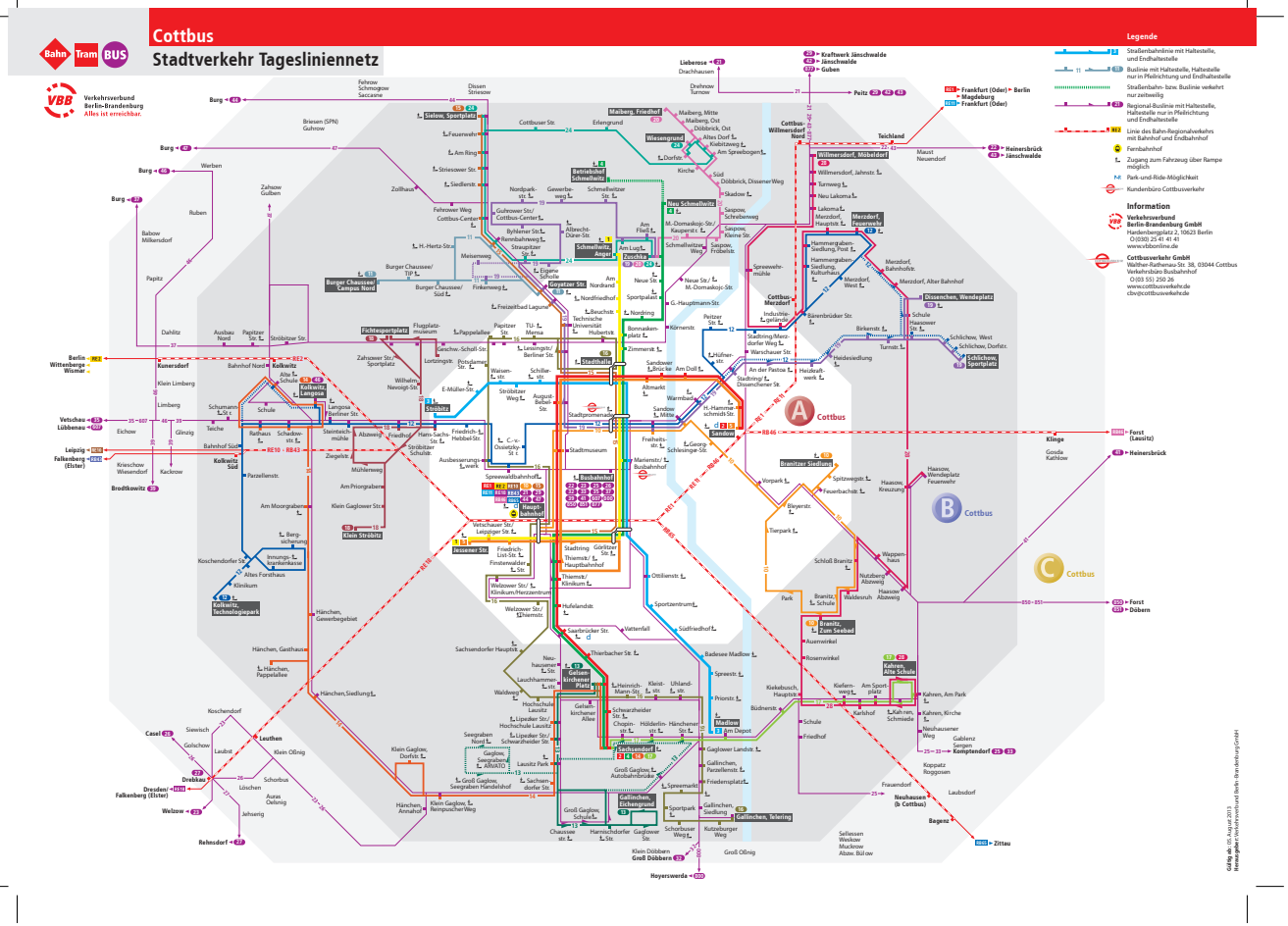


Quelle: Google Maps

$\Longrightarrow$ Kürzesten Weg von *A* nach *B* finden

Netzwerk-Strukturen im Computer erfassen

Beispiel: Liniennetzplan Tram/Bus in Cottbus



Quelle: Cottbuser Verkehrsbetriebe

$\Longrightarrow$ Schnellste Verbindung von *A* nach *B* finden

Netzwerk-Strukturen im Computer erfassen

Beispiel: Freundschaftsverbindungen bei facebook

Quelle: facebook.com

„Über maximal 6 Personen sind zwei beliebige facebook-Nutzer weltweit miteinander verbunden.“

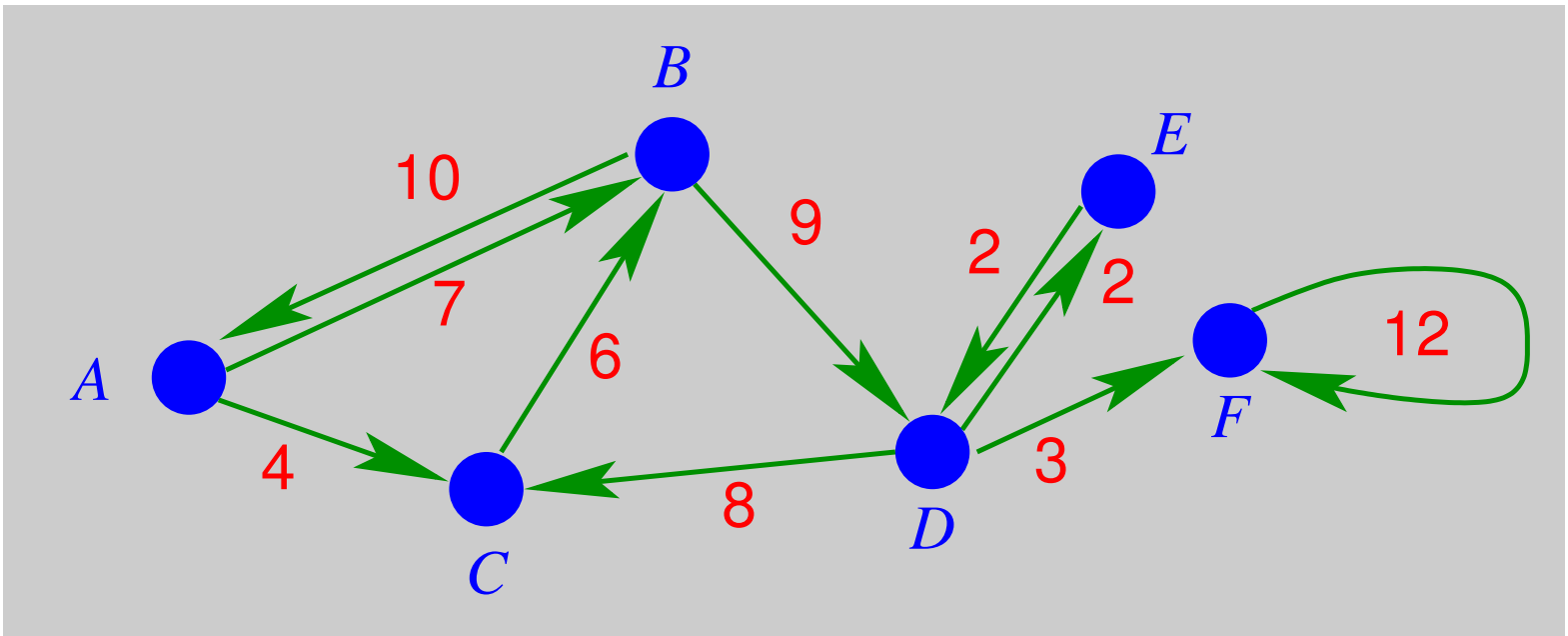
Netzwerk-Strukturen im Computer erfassen

Beispiel: Chemieanlage mit Rohrleitungsnetzwerk

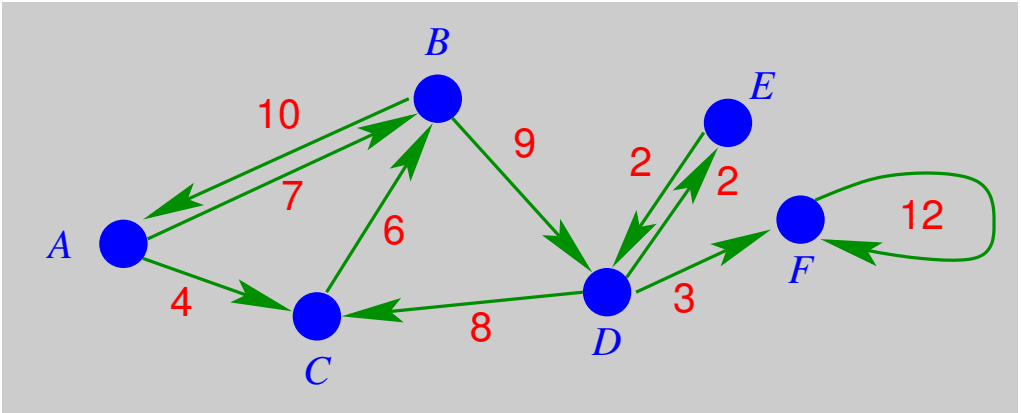
$\Longrightarrow$ Optimalen Durchfluss in der Gesamtanlage steuern, wenn eine Rohrleitung zur Reinigung abgeschaltet ist

Netzwerk als Graph abstrakt beschreiben

Ein **gerichteter Graph** ist ein Netzwerk aus *Knoten* und *Kanten*, wobei jede Kante von einem Startknoten zu einem Zielknoten führt und eine *Kantenbewertung* (z.B. Entfernung) tragen kann.



Graph durch Knoten- und Kantenmenge darstellen



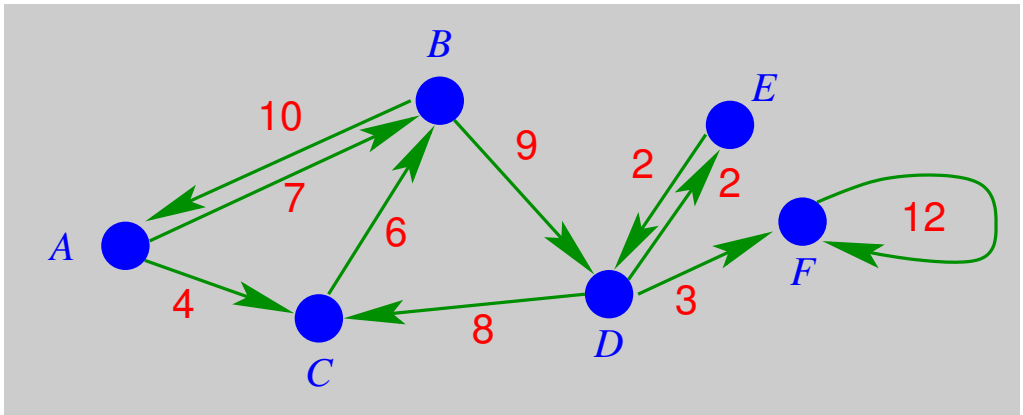
V: Knotenmenge, E: Kantenmenge

$$
\begin{aligned}
G &= (V, E) \text{ mit } E \subseteq V \times V \\
V &= \{A, B, C, D, E, F\} \\
E &= \{(A, B), (A, C), (B, A), (B, D), \\
&\quad (C, B), (D, C), (D, E), (D, F), \\
&\quad (E, D), (F, F)\}
\end{aligned}
$$

Kantenbewertungsfunktion $f : E \longrightarrow \mathbb{R}$

$$
\begin{aligned}
f(A, B) &= 7 & f(A, C) &= 4 \\
f(B, A) &= 10 & f(B, D) &= 9 \\
&\ldots & f(F, F) &= 12
\end{aligned}
$$

Graph durch Knoten- und Kantenmenge darstellen



V: Knotenmenge, E: Kantenmenge

$$
\begin{aligned}
G &= (V, E) \text{ mit } E \subseteq V \times V \\
V &= \{A, B, C, D, E, F\} \\
E &= \{(A,B),(A,C),(B,A),(B,D), \\
&\quad (C,B),(D,C),(D,E),(D,F), \\
&\quad (E,D),(F,F)\}
\end{aligned}
$$

Kantenbewertungsfunktion $f : E \longrightarrow \mathbb{R}$

$$
\begin{aligned}
f(A,B) &= 7 & f(A,C) &= 4 \\
f(B,A) &= 10 & f(B,D) &= 9 \\
&\ldots & f(F,F) &= 12
\end{aligned}
$$

Graph als Matrix
(zweidimensionales Feld)

	A	B	C	D	E	F
A	∞	7	4	∞	∞	∞
B	10	∞	∞	9	∞	∞
C	∞	6	∞	∞	∞	∞
D	∞	∞	8	∞	2	3
E	∞	∞	∞	2	∞	∞
F	∞	∞	∞	∞	∞	12

Im deutschen Flugnetz gut angebundene Flughäfen

12 Flughäfen mit jeweils mehr als einem innerdeutschen Linienziel

Entfernungstabelle innerdeutsche Direktflüge in km

	Berlin	Bremen	Dresden	Düsseldorf	FrankfurtM	Hamburg	Hannover	Köln/Bonn	Leipzig/Halle	München	Nürnberg	Stuttgart
Berlin	-			478	424			477		504	379	512
Bremen		-			320					583		479
Dresden			-	486	372	377		474		359		412
Düsseld.	478		486	-	183	339			389	487	364	
Frankf.M	424	330	372	183	-	393	262	153	293	304	187	152
Hamburg			377	339	393	-		356		612	462	534
Hannover					262		-			489		402
KölnBonn	477		474		153	356		-	380	456		
LeipzigH				389	293			380	-	360		365
München	504	583	359	487	304	612	489	456	360	-	150	191
Nürnberg	379			364	187	462				150	-	
Stuttgart	512	479	412		152	534	402		365	191		-

Entfernungstabelle innerdeutsche Direktflüge in km

	Berlin	Bremen	Dresden	Düsseldorf	FrankfurtM	Hamburg	Hannover	Köln/Bonn	Leipzig/Halle	München	Nürnberg	Stuttgart
Berlin	-			478	424			477		504	379	512
Bremen		-			320					583		479
Dresden			-	486	372	377		474		359		412
Düsseld.	478		486	-	183	339			389	487	364	
Frankf.M	424	330	372	183	-	393	262	153	293	304	187	152
Hamburg			377	339	393	-		356		612	462	534
Hannover					262		-			489		402
KölnBonn	477		474		153	356		-	380	456		
LeipzigH				389	293			380	-	360		365
München	504	583	359	487	304	612	489	456	360	-	150	191
Nürnberg	379			364	187	462				150	-	
Stuttgart	512	479	412		152	534	402		365	191		-

Von jedem Flughafen zu jedem Flughafen kürzesten Weg im Fluglinniennetz bestimmen, auch über Umstiege. Beispiel: Von Dresden nach Bremen gibt es keine Direktverbindung, aber Bremen von Dresden aus via FrankfurtM mit innerdeutschen Linienflügen auf kürzestem Weg erreichbar

Bestimmen der kürzesten Wege im Fluglinniennetz

Flugverbindungen.java – Floyd-Warshall-Algorithmus

```
public class Flugverbindungen {

  private final static int INF = 9999; //infinity: Entfernungsmasszahl wenn keine Direktverbindung

  public static void main(String[] args) {
    int z, s;
    String[] flughafen = {"Berlin", "Bremen", "Dresden", "Duesseldorf",
                          "FrankfurtM", "Hamburg", "Hannover", "KoelnBonn",
                          "LeipzigHalle", "Muenchen", "Nuernberg", "Stuttgart"};

    int entfernung[][] = { {  0, INF, INF, 478, 424, INF, INF, 477, INF, 504, 379, 512}, //Berlin
                           {INF,   0, INF, INF, 330, INF, INF, INF, INF, 583, INF, 479}, //Bremen
                           {INF, INF,   0, 486, 372, 377, INF, 474, INF, 359, INF, 412}, //Dresden
                           {478, INF, 486,   0, 183, 339, INF, INF, 389, 487, 364, INF}, //Duesseldorf
                           {424, 330, 372, 183,   0, 393, 262, 153, 293, 304, 187, 152}, //FrankfurtM
                           {INF, INF, 377, 339, 393,   0, INF, 356, INF, 612, 462, 534}, //Hamburg
                           {INF, INF, INF, INF, 262, INF,   0, INF, INF, 489, INF, 402}, //Hannover
                           {477, INF, 474, INF, 153, 356, INF,   0, 380, 456, INF, INF}, //KoelnBonn
                           {INF, INF, INF, 389, 293, INF, INF, 380,   0, 360, INF, 365}, //LeipzigHalle
                           {504, 583, 359, 487, 304, 612, 489, 456, 360,   0, 150, 191}, //Muenchen
                           {379, INF, INF, 364, 187, 462, INF, INF, INF, 150,   0, INF}, //Nuernberg
                           {512, 479, 412, INF, 152, 534, 402, INF, 365, 191, INF,   0}  //Stuttgart
                         };

    findeKuerzesteVerbindung(entfernung); //Aktualisieren der Entfernungsmatrix mit den Minimalwerten
```

Initialisierung der Entfernungsmatrix und des Flughafennamensfeldes, Methodenaufruf **`findeKuerzesteVerbindung`** zur Optimierung

Bestimmen der kürzesten Wege im Fluglinniennetz

Flughäfen `Berlin` ... `Stuttgart` per Indexwert 0 ... 11 durchnummeriert

```
private static void findeKuerzesteVerbindung(int[][] km)  //Floyd-Warshall-Algorithmus
{
  int i, j, k;
  int a;

  for (k = 0; k < km.length; k++)
  {
    for (i = 0; i < km.length; i++)
    {
      for(j = 0; j < km.length; j++)
      {
        a = km[i][k] + km[k][j];
        if (a < km[i][j])
        {
          km[i][j] = a;
        } // if
      } // for j
    } // for i
  } //for k
} // findeKuerzesteVerbindung
```

Alle Kanten $i \rightarrow j$ durchlaufen sowie alle Umwege über jeden Zwischenknoten k betrachtet. Ist die Entfernung von i nach j über k kürzer als die Direktverbindung, so wird die Matrix aktualisiert, also die eingetragene Entfernung durch die kürzere Entfernung ersetzt.

Bestimmen der kürzesten Wege im Fluglinniennetz

Schleifenvariablen `z`: Zeile, `s`: Spalte durchlaufen die Felder

```
findeKuerzesteVerbindung(entfernung); //Aktualisieren der Entfernungsmatrix mit den Minimalwerten

System.out.printf("\n             |"); //Ausgabe Tabellenueberschrift
for (z = 0; z < flughafen.length; z++)
{
  for (s = 0; s < 5; s++)
  {
    System.out.printf("%c", flughafen[z].charAt(s));
  }
  System.out.printf("|");
}
System.out.println();
for (z = 0; z < flughafen.length; z++)  //formatierte Ausgabe optimierte Entfernungsmatrix
{
  System.out.printf("%-13s|", flughafen[z]);
  for (s = 0; s < entfernung[0].length; s++)
  {
    System.out.printf("%4d |", entfernung[z][s]);
  }
  System.out.println();
}
} // main
```

	Berli	Breme	Dresd	Duess	Frank	Hambu	Hanno	Koeln	Leipz	Muenc	Nuern	Stutt
Berlin	0	754	796	478	424	817	686	477	717	504	379	512
Bremen	754	0	702	513	330	723	592	483	623	583	517	479
Dresden	796	702	0	486	372	377	634	474	665	359	509	412
Duesseldorf	478	513	486	0	183	339	445	336	389	487	364	335
FrankfurtM	424	330	372	183	0	393	262	153	293	304	187	152
Hamburg	817	723	377	339	393	0	655	356	686	612	462	534
Hannover	686	592	634	445	262	655	0	415	555	489	449	402
KoelnBonn	477	483	474	336	153	356	415	0	380	456	340	305
LeipzigHalle	717	623	665	389	293	686	555	380	0	360	480	365
Muenchen	504	583	359	487	304	612	489	456	360	0	150	191
Nuernberg	379	517	509	364	187	462	449	340	480	150	0	339
Stuttgart	512	479	412	335	152	534	402	305	365	191	339	0

Formatierte Ausgabe der optimierten Entfernungsmatrix im zweiten Teil der **`main`**-Methode