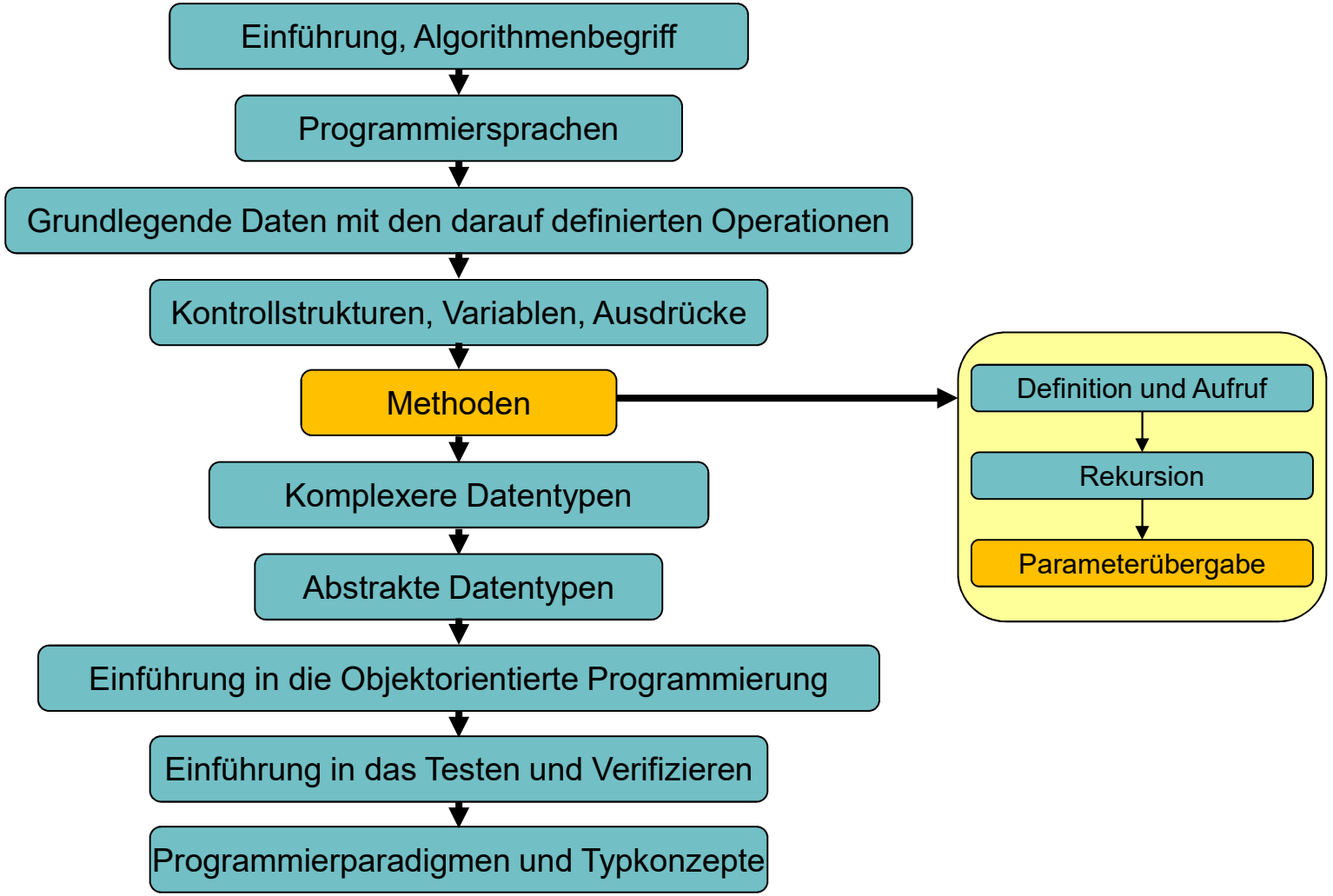
Inhalt dieser Veranstaltung

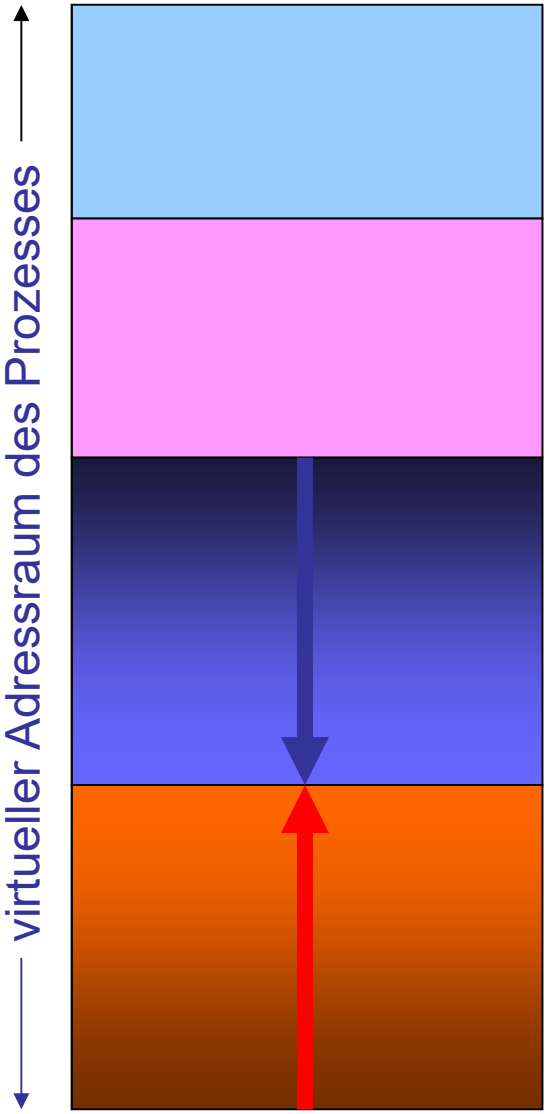


Methodenaufruf

```
public class Abstand {
    public static void main (String[] args) {
        double x=0.0, y=0.0;
        System.out.println("Abstand Punkte ist " + abstand(x, y, x+1, y+3));
    }

    public static double abstand (double x1, double y1, double x2, double y2) {
        return Math.sqrt(quadrat(x2-x1) + quadrat(y2-y1));
    }

    public static double quadrat(double x) {
        return x*x;
    }
}
```

- Wiederholung:
  - Auswerten der Ausdrücke der aktuellen Parameter von links nach rechts
  - Belegung der formalen Parameter mit diesen Werten
  - Ausführung des Codes der Methode
  - Ermittlung des Ergebniswertes (durch return-Anweisung angegeben)
  - "Ersetzen" des Methodenaufrufs durch diesen Wert
- Aber was passiert da genau intern? Jetzt die Prinzipien der internen Arbeitsweise...

Speicheraufteilung für ein Programm



1. Im Code-Bereich wird der Maschinencode des Programms abgelegt. Keine Größenveränderungen zur Laufzeit.

2. Im statischen Datenbereich werden Daten abgelegt, die zu Beginn des Programmlaufs angelegt werden und bis zum Programmende existieren. Keine Größenveränderung.

3. Auf dem Laufzeitstack werden dynamische Aufrufbereiche von Methodenaufrufen abgelegt. Dazu gehören auch (nicht-statische) lokale Variablen von Methoden. Die Größe des Laufzeitstacks ist dynamisch, er wächst und schrumpft nach Bedarf.

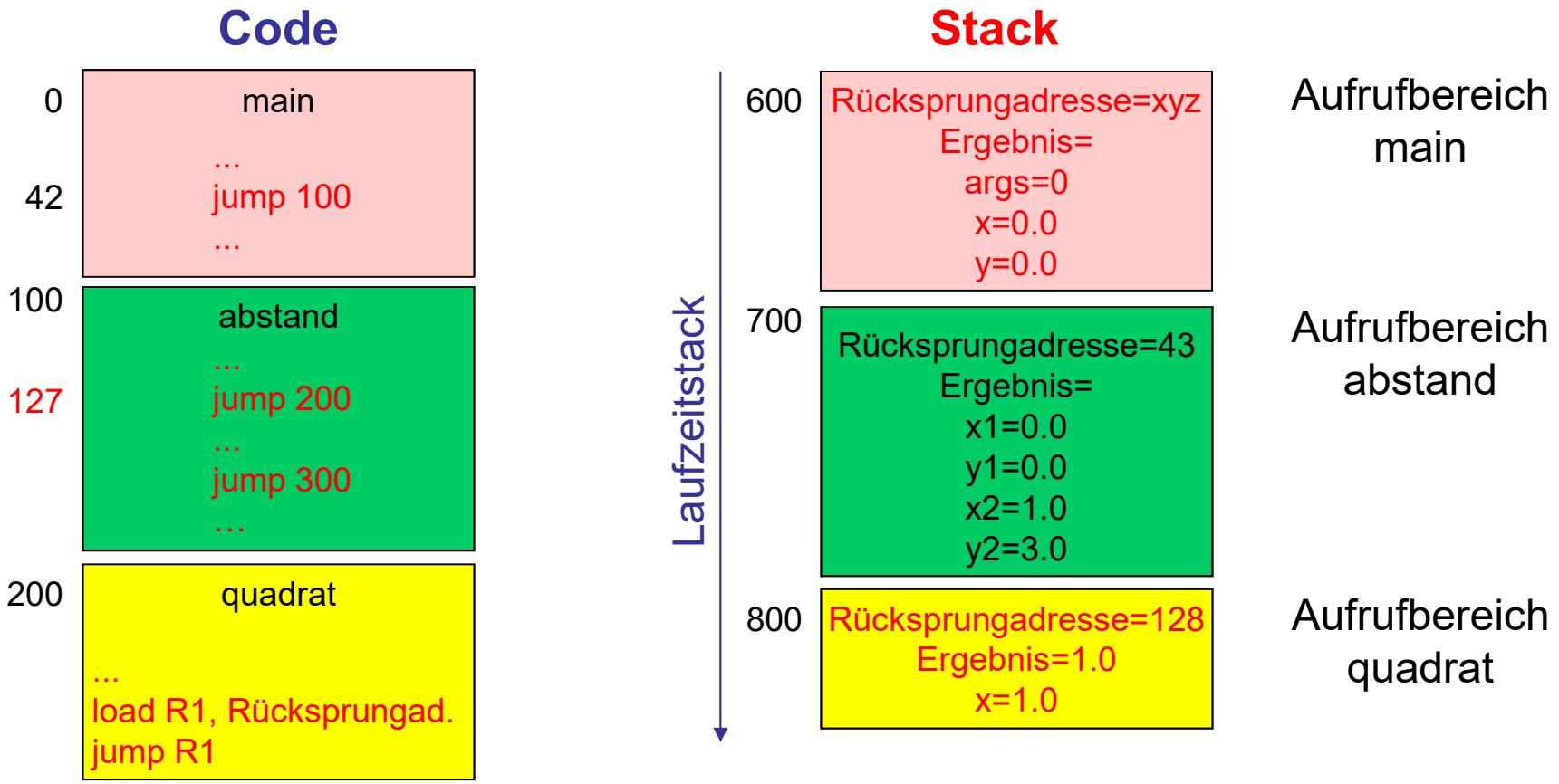
4. Der Heap nimmt Datenobjekte auf, die zur Laufzeit neu angelegt werden. Wir werden später solche Objekte kennen lernen. Die Größe des Heaps ändert sich ebenfalls dynamisch.

Aufrufbereiche

- Für Methodenaufrufe müssen Informationen / Daten zwischen der aufrufenden und der aufgerufenen Methode übergeben werden
- Dazu dienen Aufrufbereiche, die auf dem Laufzeitstack im Hauptspeicher angelegt werden und über die in einem festen Format die Daten übermittelt werden
- Zu jedem Methodenaufruf gibt es einen Aufrufbereich, der solange auf dem Laufzeitstack liegt, wie der Methodenaufruf noch in Bearbeitung ist
- Der Laufzeitstack wächst damit, wenn innerhalb eines Methodenaufrufs weitere Methodenaufrufe stattfinden (zum Beispiel bei Rekursion)

- In einem Aufrufbereich sind angelegt (vom Aufrufer bzw. Aufgerufenen mit Werten belegt):
  - Rücksprungadresse der Codestelle, an der in der aufrufenden Methode die Ausführung später fortgesetzt werden soll
  - der Resultatwert (bei void-Funktionen bleibt dieser leer; oft auch in einem festen Resultatregister übergeben)
  - Werte der Übergabeparameter
  - Alle lokalen Variablen der Methode

Beispiel zum Stand Ende quadrat(x2-x1)

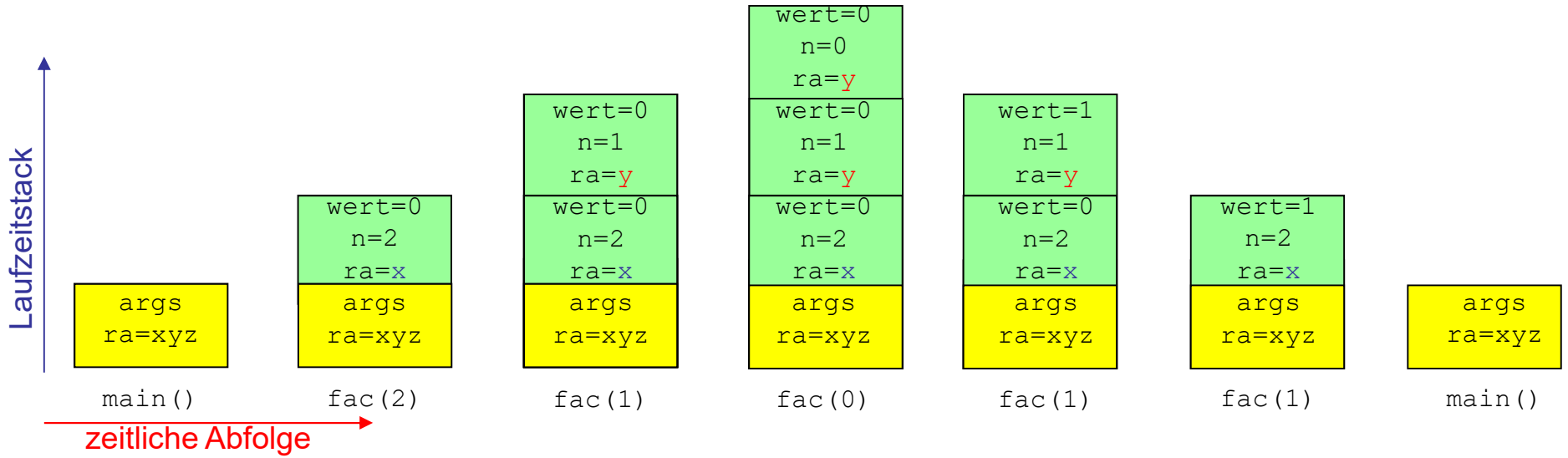


Hinweis:

- Darstellung nicht vollständig
- nur einige der für die Diskussion relevanten Aspekte zu einem Zeitpunkt der Auswertung sind angegeben

Beispiel zum zeitlichen Ablauf

```
public class Fakultaet {
    public static int fac(int n) {
        int wert = 0;
        if(n == 0) return 1;
        else {
            wert = fac(n-1);        // Position y
            return n*wert;
        }
    }
    public static void main(String[] args) {
        System.out.println("fakultaet(2)=" + fac(2));    // Position x
    }
}
```



Zwischenstand

- Es existieren für Daten drei Speicherbereiche, die gewisse Eigenschaften haben (insbesondere wie sie verwaltet werden,…)
- Programmiersprachen enthalten Konstrukte, um eine Aufteilung von Daten / Variablen auf diese Bereiche zu beeinflussen
- Mit jedem Methodenaufruf werden Verwaltungsinformationen auf dem Laufzeitstack abgelegt

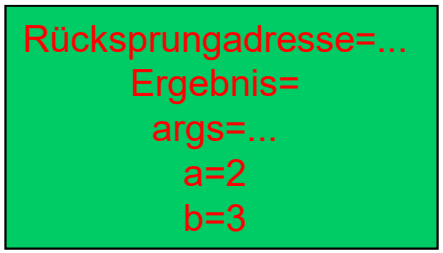
Reflektion

- Gibt es Programme, für die man eine maximale Stack-Größe angeben kann?
- Könnte man für jedes Programm eine maximale Stack-Größe angeben?
- Könnte man eine Stack-Größe angeben, die für alle Programme reichen würde?

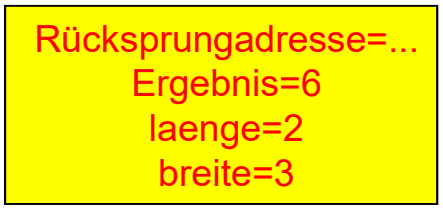
call-by-value

```
public class Uebergabestrategien {
    public static void main(String[] args) {
        int a=2, b=3;
        System.out.println("Die Fläche vom Rechteck ist " + flaeche(a,b));
    }

    public static int flaeche(int laenge, int breite) {
        return laenge * breite;
    }
}
```

Aufrufbereich von main:



Aufrufbereich von flaeche:



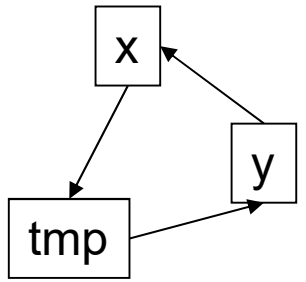
- Werden die Werte der aktuellen Parameter (in Kopie) übergeben, so spricht man von einer call-by-value Übergabestrategie
- Java nutzt diese call-by-value Übergabestrategie

swap

Aufgabe: Tausche den Inhalt zweier Variablen über eine Methode

```
public class Uebergabestrategien2 {
    public static void main(String[] args) {
        int a=2, b=3;
        System.out.println("vorher: a=" + a + ", b=" + b);
        swap(a,b);
        System.out.println("nachher: a=" + a + ", b=" + b);
    }

    public static void swap(int x, int y) {
        int tmp = x
        x = y;
        y = tmp;
        System.out.println("in swap: a=" + x + ", b=" + y);
    }
}
```

Dreieckstausch



Ausgabe:

```
vorher: a=2, b=3
in swap: a=3, b=2
nachher: a=2, b=3
```

Hier passiert also **nicht** das Gewünschte!
Die Kopien werden getauscht, aber nicht die Originalwerte.
Grund: call-by-value

swap in Fortran

- Das swap-Programm in der Programmiersprache Fortran (Formula Translator):

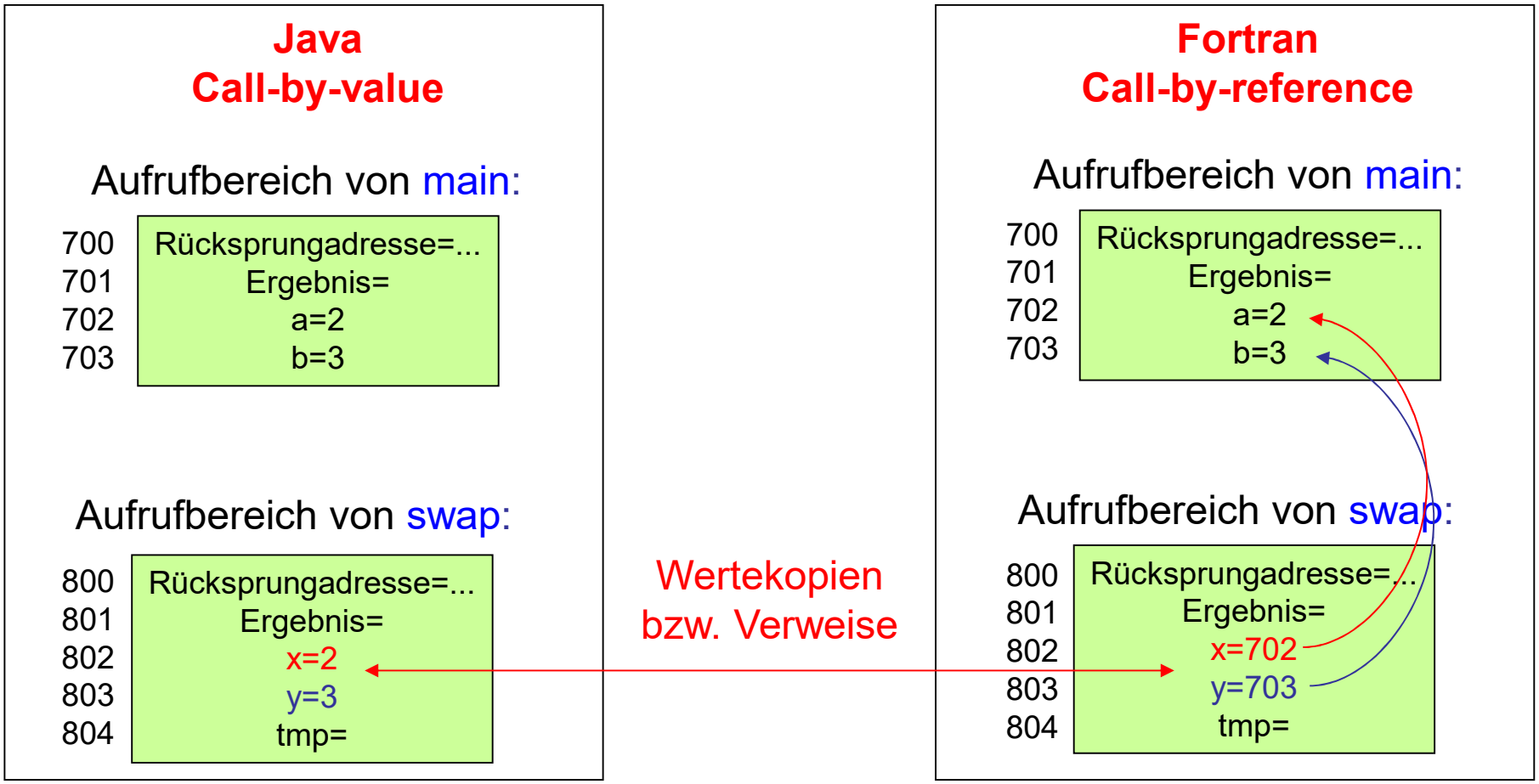
```
program main
  integer a,b
  a = 2
  b = 3
  print a,b
  call swap(a,b)
  print a,b
end
```

```
subroutine swap(x,y)
  integer x,y
  integer tmp
  tmp = x
  x = y
  y = tmp
  return
end
```

Ausgabe (wie gewünscht):
2 3
3 2

- Grund: Fortran wendet die call-by-reference Übergabestrategie an
- Dabei werden anstatt der Werte die Adressen (Verweise, Zeiger) übergeben, wo die Werte zu finden sind
- Dadurch werden die Originalinhalte vertauscht, und nicht Wertkopien
- Bei der Nutzung von formalen Parametern in der aufgerufenen Methode generiert der Compiler automatisch Code, der statt mit den Werten der Variablen (=Zeiger) über die Zeiger mit den Originalwerten arbeitet

Unterschied call-by-value / call-by-reference



Zwischenstand

- Call-by-value und call-by-reference sind Strategien der Parameterübergabe mit unterschiedlicher Auswirkung

Reflektion

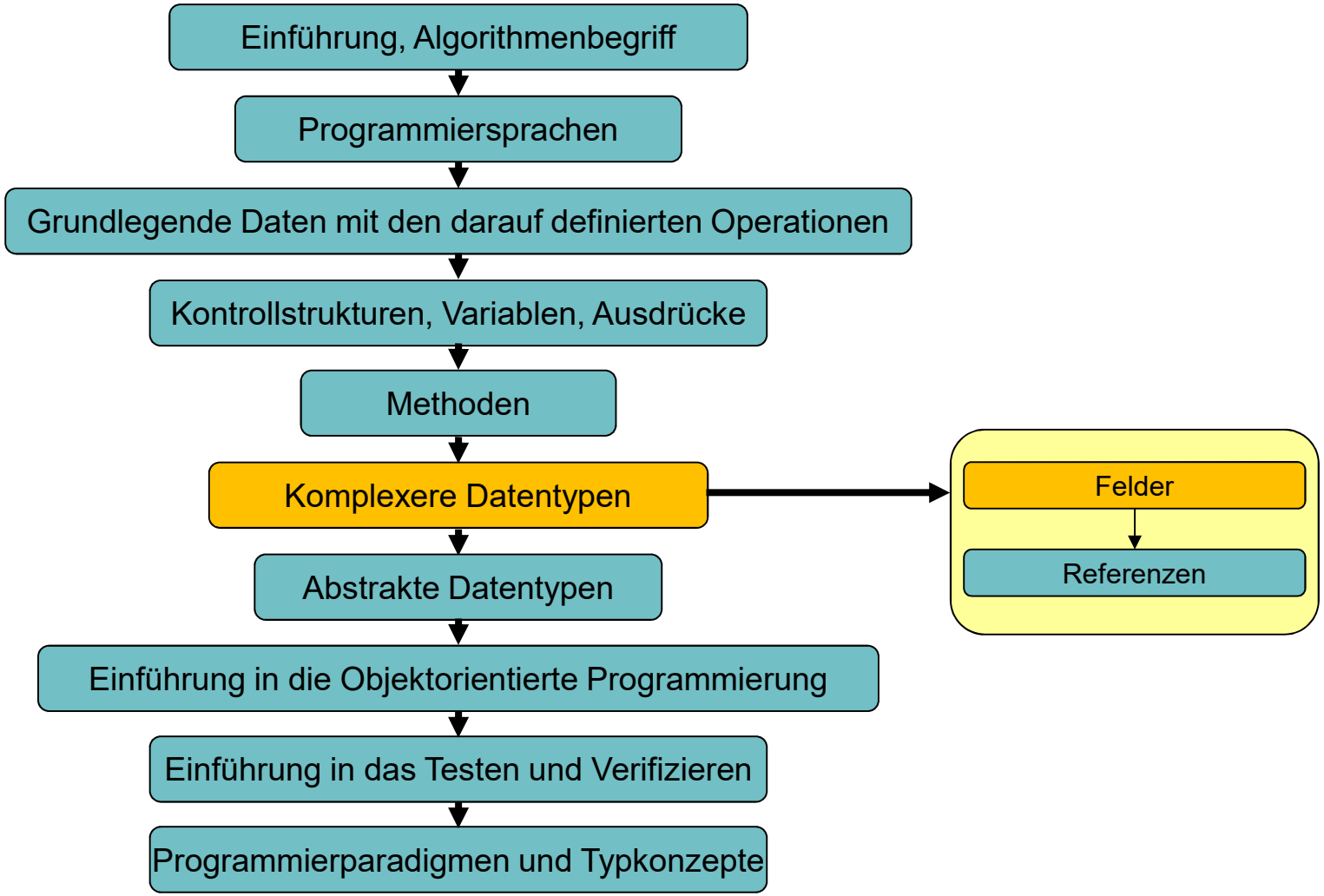
- Sind dies jetzt alle Möglichkeiten zur Übergabe von Parameterwerten oder könnte man sich auch weitere Übergabestrategien überlegen? Wenn nein, wieso nicht? Wenn ja, wie könnte beispielsweise eine weitere Strategie aussehen?

Zusammenfassung

- Aufrufbereiche dienen der Übergabe von Informationen zwischen aufrufender und aufgerufenen Methode in einem festgelegten Format
- Aufrufbereiche werden auf dem Laufzeitstack angelegt (und wieder nach Beendigung der Aufrufbearbeitung wieder gelöscht
- Der Laufzeitstack ist begrenzt, viele offene Methodenaufrufe (mit vielen lokalen Daten einer Methode) können ein Problem werden

- Java nutzt die call-by-value Übergabestrategie
- Dadurch können Originalwerte von Variablen der aufrufenden Methode (zum jetzigen Zeitpunkt) nicht geändert werden

Inhalt dieser Veranstaltung



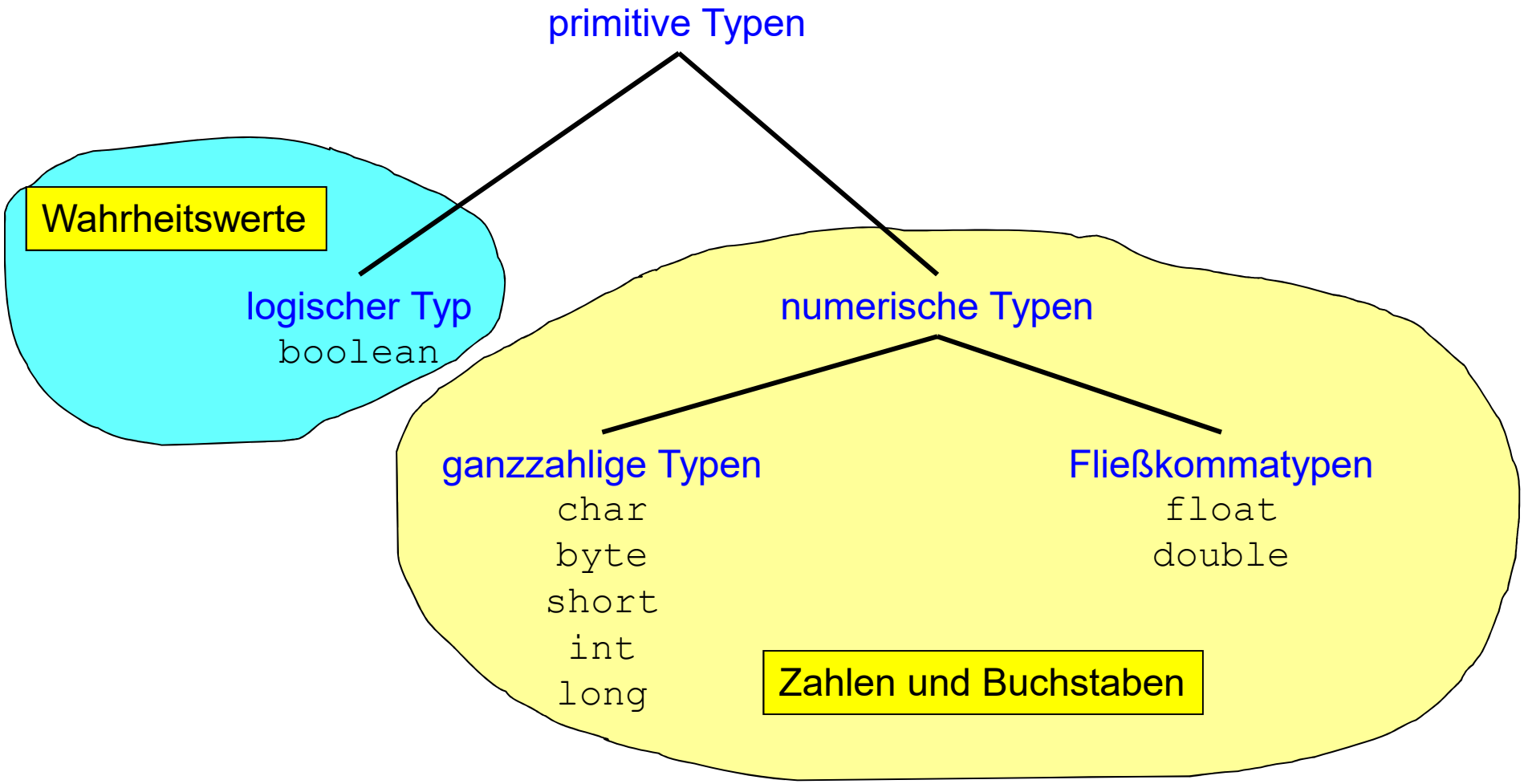
Reichen einzelne Werte?

- Werte, die wir bis jetzt in Java kennen: skalare Werte

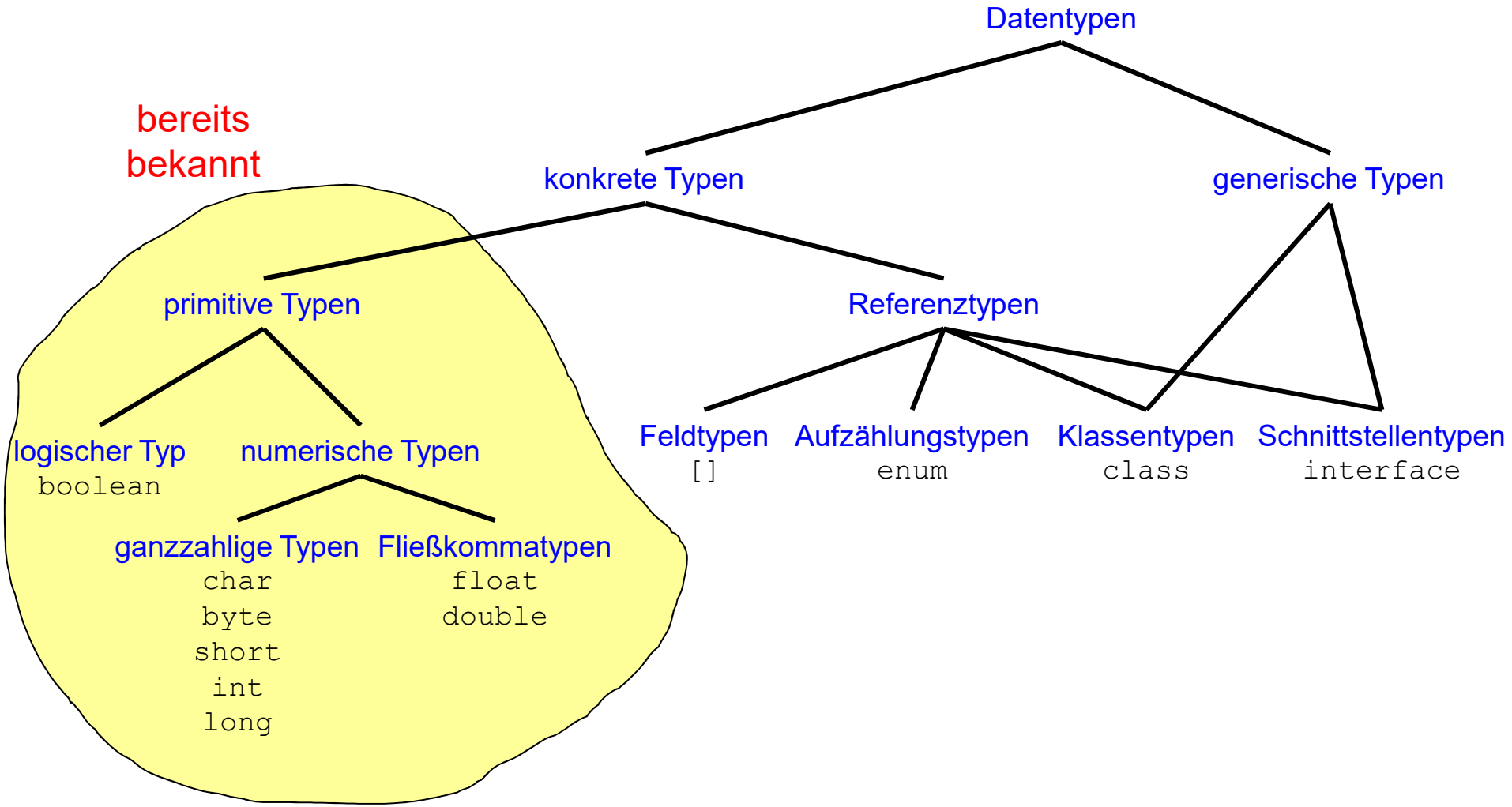
- Zum Beispiel aus der Mathematik sind aber auch bekannt:
  - Mengen: $\{x_1,...,x_n\}$, N, $\{ x \mid x \in N, x>10\}$
  - Folgen: (a_i)
  - Kartesisches Produkt: $A \times B \times C$ für Mengen A,B,C
  - Vektoren: $(x_1,...,x_n) \in R^n$
  - Matrizen: $\begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix} \in R^{n,m}$
  - ...

- Dies sind Behälter für die eigentlichen Daten
- Diese Behälter haben bestimmte Eigenschaften und es sind bestimmte Operationen darauf definiert
- Beispiel:
  - Menge: Reihenfolge der Elemente irrelevant; Vereinigung, Schnitt,...
  - Vektor: Reihenfolge ist relevant; Addition, Skalarprodukt,...

Bis jetzt bekannte Datentypen in Java



Aussicht: Das Java Typuniversum



Feld / Vektor

- Felder / Vektoren spielen eine wichtige Rolle in vielen Anwendungsbereichen
- Feld/Vektor $(x_1, ..., x_n) \in R^n$: feste Anzahl von Datenelementen eines homogenen Basistyps (im Beispiel R)

- Wichtige Operation: direkter Zugriff auf das i-te Element durch Indizierung x_i mit $i \in \{1,...,n\}$ möglich

- Beispiel: Mittelwert aller inneren Elemente eines Vektors mit den beiden direkten Nachbarn:

$$(x_{i-1} + x_i + x_{i+1}) / 3 \quad \forall\ i=2,...,n-1$$

Formel allgemein ausgedrückt

Wiederholungsvorschrift für Bezugselemente

Felddatentyp in Java

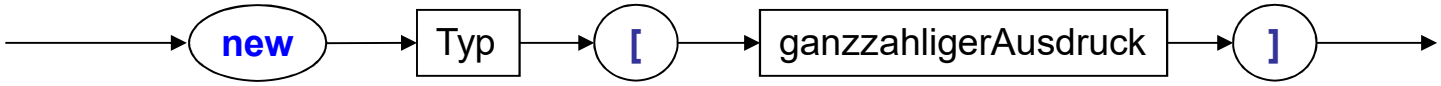
- Zu einem beliebigen Java-Typ `T` kann man einen Feldtyp angeben: `T[]`
- Beispiele:
  - `int[]` : Typ „Feld von int-Werten“
  - `float[]` : Typ „Feld von float-Werten“
  - `char[]` : Typ „Feld von char-Werten“
  - `String[]` : Typ „Feld von String-Werten“
- `T[]` ist wieder ein Java-Typ
- Überall dort, wo ein Typ verlangt wird, kann man auch einen Feldtyp angeben
- Beispiele: Variablendeklaration, Ergebnistyp einer Methode, Parametertypen einer Methode

```
int[] a;
public static int[] methode(float[] a1, byte[] a2) {...}
```

- Mit einem Feldtyp in Java ist nicht festgelegt, wie viele Elemente ein Feld dieses Typs hat. In einem konkreten Feld**wert** wird gleich die Anzahl festgelegt, aber nicht in einem Feld**typ**!

Anlegen eines (eindimensionalen) Feldes in Java

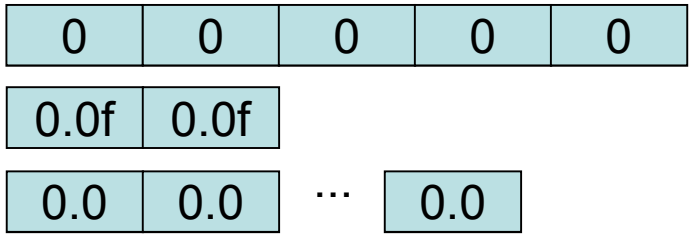
Felderzeugung



- Zu einem Basistyp `T` kann man ein Feld, also einen Wert des entsprechenden Feldtyps erzeugen mit

  ```
  new T[anzahl]
  ```

  wobei der ganzzahlige Ausdruck `anzahl` in den eckigen Klammern konkret die Anzahl der Elemente des neu zu erzeugenden Feldes angibt
- Die Feldelemente enthalten zu Beginn Null-Elemente des Basisdatentyps
- Beispiele (für eine int-Variable `n`):
  - `new int[5]`
  - `new float[2]`
  - `new double[3*n]`

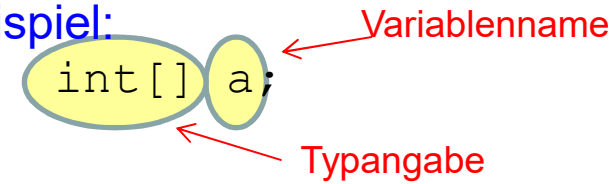


- Der Speicherbedarf für das Gesamtfeld ist ca.

  Anzahl Elemente · Speicherbedarf(Basistyp)
- Die Erzeugung eines Feldes ist ein Ausdruck, liefert also einen Wert (gleich mehr dazu)

Feldvariablen

- Analog zu den primitiven Datentypen kann man auch zu einem Feldtyp eine Variable dieses Typs anlegen
- Beispiel:



- Die Definition einer Feldvariablen lässt sich auch mit dem Erzeugen eines Feldes kombinieren (siehe Initialisierungsausdruck früher)
- Beispiel:

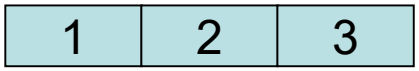
```
int[] a = new int[3];
```

- Wir werden gleich aber noch einen wichtigen Unterschied zwischen Variablen eines primitiven Typs und Variablen eines Referenztyps sehen!
- Der Wert einer Feldvariablen (in einem Ausdruck) ist das Feld selber

Implizites Anlegen eines Feldes

- Neben dem Anlegen eines Feldes mit den `new`-Operator gibt es weiterhin die Möglichkeit des impliziten Anlegens durch einen Initialisierungsausdruck bei der Deklaration einer Variablen von einem Feldtyp (und nur dort)
- Beispiel:

```
int[] a = {1,1+1,3};
```



- Das Feld wird mit so vielen Feldelementen angelegt, wie es Initialisierungsausdrücke in den geschweiften Klammern gibt
- Die einzelnen Feldelemente werden mit den entsprechenden Werten vorbelegt

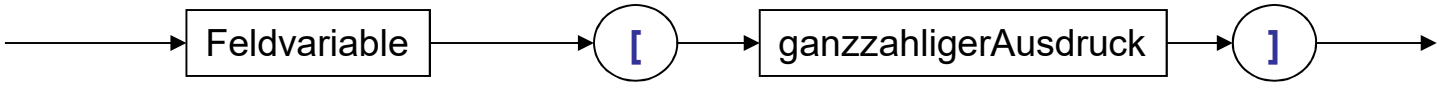
- Weiterführende Bemerkung: Weiterhin kann auch ein anonymes Feld (ohne Variable) angelegt werden:

```
new int[]{1,2,3}
```

Länge und Indizierung von Feldern

- Zu jedem Feld mit einer Feldvariablen `a` kann man die Anzahl der Elemente erfragen: `a.length` (ein ganzzahliger Ausdruck)

Feldindizierung



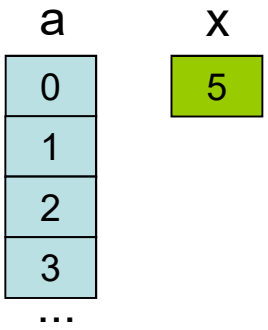
- Der Zugriff auf ein Feldelement geschieht durch Indizierung des Feldes mit eckigen Klammern. Dies entspricht in der Mathematik der Schreibweise a_i
- Beispiel:

```
a[i+3]
```

- Wichtig: der Index des ersten Feldelementes ist `0`, der des letzten ist `a.length-1` für eine Feldvariable `a`

- Die einzelnen Feldelemente werden wie Variablen des entsprechenden Basistyps behandelt
- Beispiel: `a[i] = a[i]+1;`

Beispiel: Suchen in einem Feld



$\forall$ i=1,...,n

$a_i = i$

Wiederholungs-vorschrift für Bezugselemente

$\forall$ i=1,...,n

Zusammenhang allgemein ausgedrückt

```
public class SequentiellesSuchen {
    public static void main(String[] args) {
        int x = 5;                  // zu suchender Wert
        int[] a = new int[10];   // Feld mit Daten
        // Belegung des Feldes mit Beispielwerten
        for(int i=0; i<a.length; i++) {
            a[i] = i;
        }
        // Suchen
        boolean gefunden = false;
        int position = -1;
        for(int i=0; i<a.length; i++) {
            if(a[i] == x) {
                gefunden = true;
                position = i;
                break;
            }
        }
        if(gefunden) {
            System.out.println("gefunden an Position " + position);
        } else {
            System.out.println("nicht gefunden");
        }
    }
}
```

Als Methode

Ergebnis ist Position / Index des gesuchten Wertes in dem Feld.

Falls der Wert nicht vorhanden ist, so wird -1 (ungültiger Index) als Ergebnis geliefert

```
public class SequentiellesSuchen2 {
    public static int suchen(int[] a, int x) {
        int position = -1;
        for(int i=0; i<a.length; i++) {
            if(a[i] == x) {
                position = i;
                break;    // alternativ hier mit return arbeiten
            }
        }
        return position;
    }

    public static void main(String[] args) {
        int x = 5;                          // zu suchender Wert
        int[] a = {0,1,2,3,4,5,6,7,8,9};   // Feld mit Daten

        // suchen
        int position = suchen(a, x);
        if(position >= 0) {
            System.out.println("gefunden an Pos. " + position);
        } else {
            System.out.println("nicht gefunden");
        }
    }
}
```

Potentielle Probleme mit Feldern

- Speicherbedarf zu groß

```
public class VielSpeicher {
    public static void main(String[] args) {
        int[] feld = new int [1000000000];
    }
}
```

```
prompt> java VielSpeicher
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at VielSpeicher.main(VielSpeicher.java:3)
```

- Indizierung fehlerhaft

```
public class Indexverletzung {
    public static void main(String[] args) {
        int[] feld =  new int [2];
        feld[2] = 4;
    }
}
```

```
> java Indexverletzung
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2
        at Indexverletzung.main(Indexverletzung.java:4)
```

Anwendung: Sieb des Eratosthenes

- Zur Erinnerung: Eine Primzahl ist nur durch 1 und sich selber teilbar.
- Mit dem Verfahren "Sieb des Eratosthenes" lassen sich alle Primzahlen bis zu einer vorgegebenen Zahl N berechnen.

- Das Verfahren beruht auf den folgenden Grundgedanken:
  - In einer Menge (dem Sieb) sind zu Anfang alle Zahlen 2,...,N enthalten.
  - Man nimmt sich die nächst kleinste noch nicht betrachtete Zahl aus der Menge. Diese Zahl ist eine Primzahl.
  - Man streicht alle Vielfachen dieser Zahl aus der Menge.
  - Diese Schritte wiederholt man, solange noch nicht betrachtete Zahlen im Sieb vorhanden sind.

Beispiel

N=10:

Betrachtete Zahl	Sieb	Vielfache	Sieb - Vielfache
2	{2,3,4,5,6,7,8,9,10}	{2,4,6,8,10}	{3,5,7,9}
3	{3,5,7,9}	{3,6,9}	{5,7}
5	{5,7}	{5,10}	{7}
7	{7}	{7}	{}

Umsetzung in ein Programm

- Wir müssen uns Gedanken machen darüber
  - wie die Daten abgebildet werden
  - wie der Kontrollfluss abgebildet wird

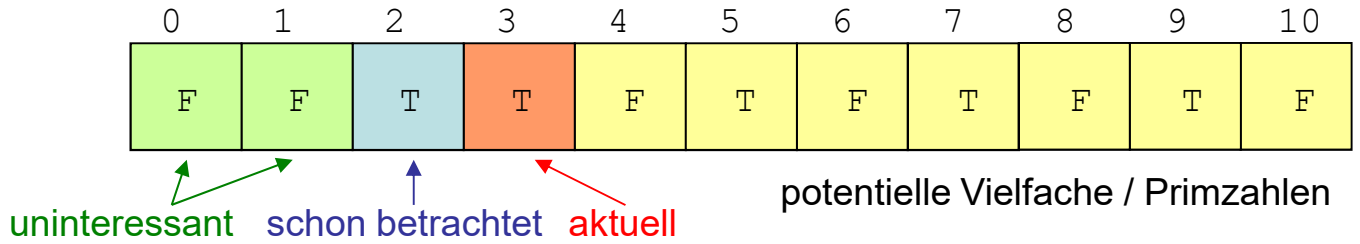
- Lösung:
  - Darstellung: Das Sieb ist ein Feld der Dimension N+1 von boolschen Werten. Ein Wert `i` ist in der Menge enthalten (ist im Sieb), wenn `feld[i]` gleich `true` ist.
  - Kontrollfluss: Der Einfachheit halber macht man eine Schleife von 2,...,N und schaut für jeden Wert `i`, ob `feld[i]` den Wert `true` hat (ist Primzahl) oder nicht. Das Abbruchkriterium ist damit das Ende das Schleife.
  - (Überlegen Sie sich, ob eine Schleifenobergrenze N wirklich notwendig ist.)

Beispiel:

N=10

betrachtete Zahl=3

restliches Sieb={3,5,7,9}



Java-Programm

```
public class Eratosthenes
{
    public static void main(String[] args) {
        // bis zu welcher Zahl wollen wir die Primzahlen ausrechnen?
        int maximalzahl = Integer.parseInt(args[0]);

        // Das Sieb (Achtung: Indizes laufen von 0..n-1 in Java, deswegen +1)
        boolean[] sieb = new boolean[maximalzahl+1];

        // Setze alle relevanten Werte im Sieb auf wahr
        for( int i = 2; i < maximalzahl+1; i++)
            sieb[i] = true;

        // Fuehre den folgenden Test fuer alle Zahlen ab 2 durch
        for(int startzahl = 2; startzahl < maximalzahl+1; startzahl++) {
            if(sieb[startzahl])  {
                // Primzahl gefunden: Gebe Information aus
                System.out.println(startzahl + " ist eine Primzahl");

                // Streiche alle Vielfachen im Sieb
                for(int vielfaches=2*startzahl; vielfaches<=maximalzahl;vielfaches+=startzahl)
                    sieb[vielfaches] = false;
            }
        }
    }
}
```

Mehrdimensionale Felder

- Zur Erinnerung: zu einem Typ `T` ist `T[]` wieder ein Java-Typ
- Jetzt: Basistyp ist `T[]`, abgeleiteter Feldtyp dazu: `T[][]`
- In Java ist ein d-dimensionales Feld ein Feld von (d-1) dimensionalen Feldern (siehe Typherleitung eben)

- Schreibweise Mathematik (d=2):
  - Matrix: $R^{n \times m}$
  - Ein Matrixelement: $a_{i,j}$

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}$$

- Schreibweise Java (d=2):
  - Matrix: `double[][] a = new double[n][m];`
  - Ein Matrixelement: `a[i][j]`
  - Anzahl Zeilen: `a.length`
  - Anzahl Spalten: `a[0].length` (=Anzahl Elemente in Zeile 0)

Beispiel: Addition zweier gleichgroßer Matrizen

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} + \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix}$$

```
public class MatrixAddition {

    public static double[][] addieren(double[][] a, double[][] b) {
        // Achtung: wir ueberpruefen hier nicht,
        // ob beide Matrizen gleich gross sind!
        // Ergebnismatrix erzeugen
        double[][] c = new double[a.length][a[0].length];

        for(int i=0; i<a.length; i++) {
            for(int j=0; j<a[i].length; j++) {
                c[i][j] = a[i][j] + b[i][j];
            }
        }
        return c;
    }

    public static void main(String[] args) {
        double[][] a = {{1,2,3}, {4,5,6}, {7,8,9}};
        double[][] b = {{9,8,7}, {6,5,4}, {3,2,1}};

        double[][] c = addieren(a,b);
    }
}
```

Lösungsansatz: Rekursion

- Gegeben sind n Werte in einer vorgegebene Reihenfolge, zum Beispiel in Form eines Feldes `a=[1,2,3,4]`
- Aufgabe: Erzeuge systematisch alle Permutation dieser Werte `[1,2,3,4]`, `[1,3,2,4],...` Die Reihenfolge der Permutationen kann beliebig sein.

- Strategie:
  Um für n Elemente alle Permutationen zu erzeugen, vertausche der Reihe nach das i-te Element (i zwischen 0 und n-1) mit dem letzten Element, erzeuge damit alle Permutationen der Länge n-1 (ohne letztes Element) und tausche anschließend wieder das i-te Element zurück

- Beispiel: gegeben `[1,2,3]`, n=3
  - i=0: vertauschen liefert `[3,2,1]`, erzeuge rekursiv Permutationen von `[3,2]`
  - i=1: vertauschen liefert `[1,3,2]`, erzeuge rekursiv Permutationen von `[1,3]`
  - i=2: vertauschen liefert `[1,2,3]`, erzeuge rekursiv Permutationen von `[1,2]`

Programm

```
/** erzeuge alle Permutationen des Feldes
  * @args a zu permutierendes Feld
*/
public static void erzeugePermutationen(int[] a) {
     // rufe rekursive Methode auf, die die eigentliche Permutation bewirkt
     permutiere(a, a.length);
}

// erzeuge rekursiv alle Permutationen der Laenge n
private static void permutiere(int[] a, int n) {
    if(n == 1) {
        zeigeFeld(a);                // genug permutiert
    } else {
        for(int i=0; i<n; i++) {
            swap(a, i, n-1);         // tausche a[i] mit letztem Element
            permutiere(a, n-1);      // erzeuge alle Permutation der Laenge n-1
            swap(a, i, n-1);         // tausche zurueck
        }
    }
}
```

Lösungsansatz: Backtracking

- Lösungsstrategie Backtracking:
  - Gehe vom derzeitigen Stand einen noch nicht verfolgten nächsten Schritt weiter und versuche von dort aus zum Ziel zu kommen
  - Wenn dieser Schritt nicht zum Ziel führt, nehme diesen Schritt zurück und verfolge einen anderen alternativen Schritt
  - Wende diese Strategie überall an

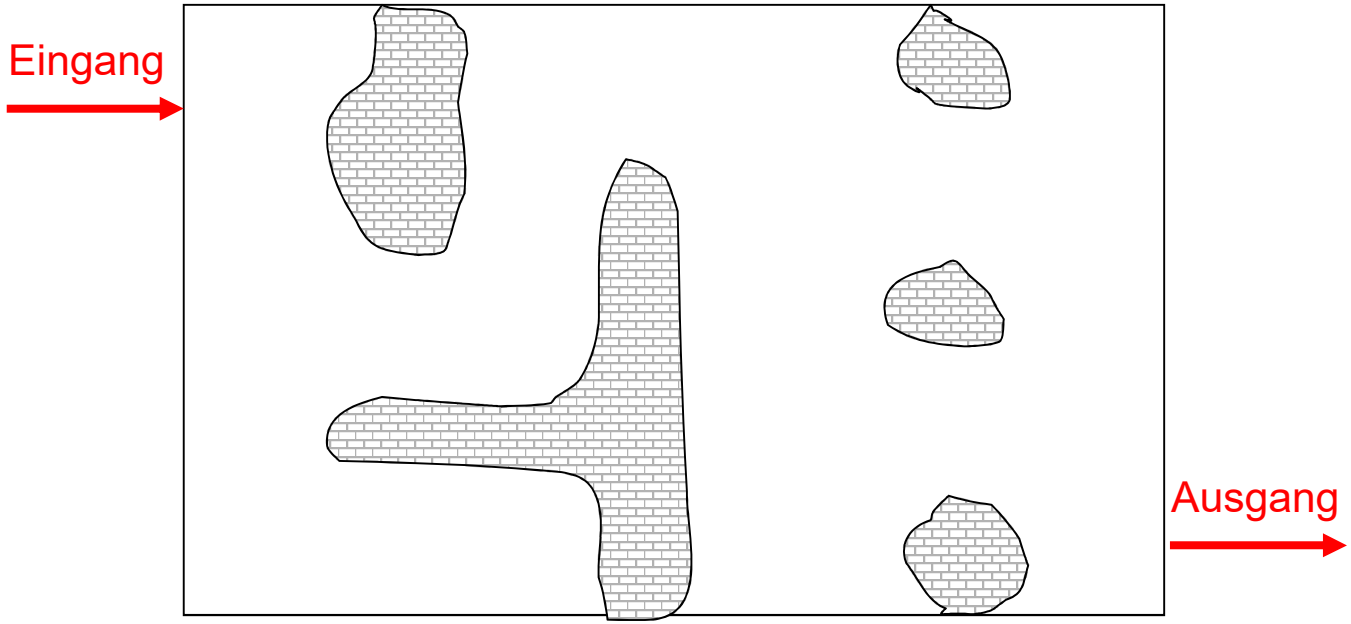
- Im schlimmsten Fall würde man mit dieser Strategie alle möglichen Fälle ausprobieren

Wegsuche in einem Labyrinth

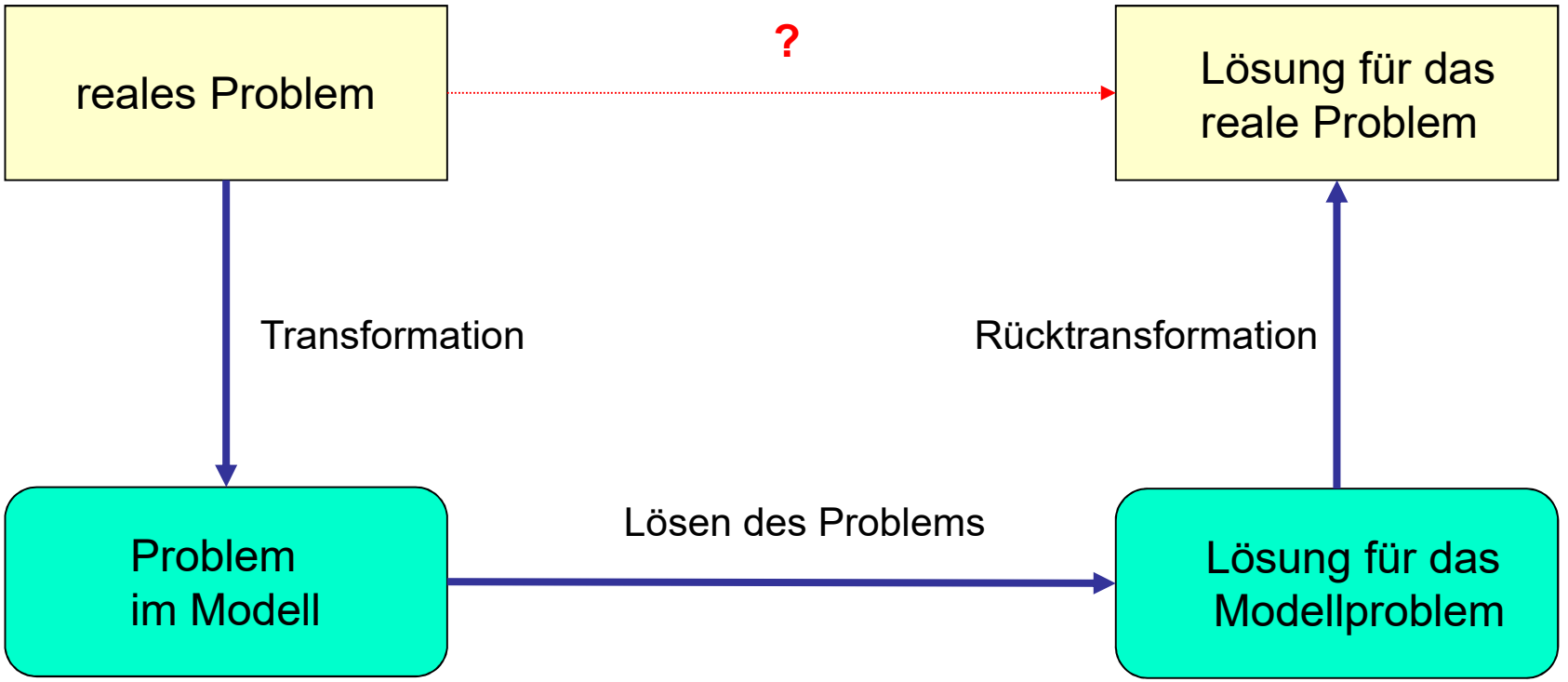
- Problem: Für ein beliebiges Labyrinth soll man einen Weg von einem Startpunkt zu einem Endpunkt durch das Labyrinth finden.

- Gesucht: Algorithmus, der das Problem allgemein löst, d.h. mit dem man einen Weg (soweit vorhanden) durch ein beliebiges Labyrinth findet.

- Hinweis: es gibt weitere Ansätze zur Lösung dieses Problems

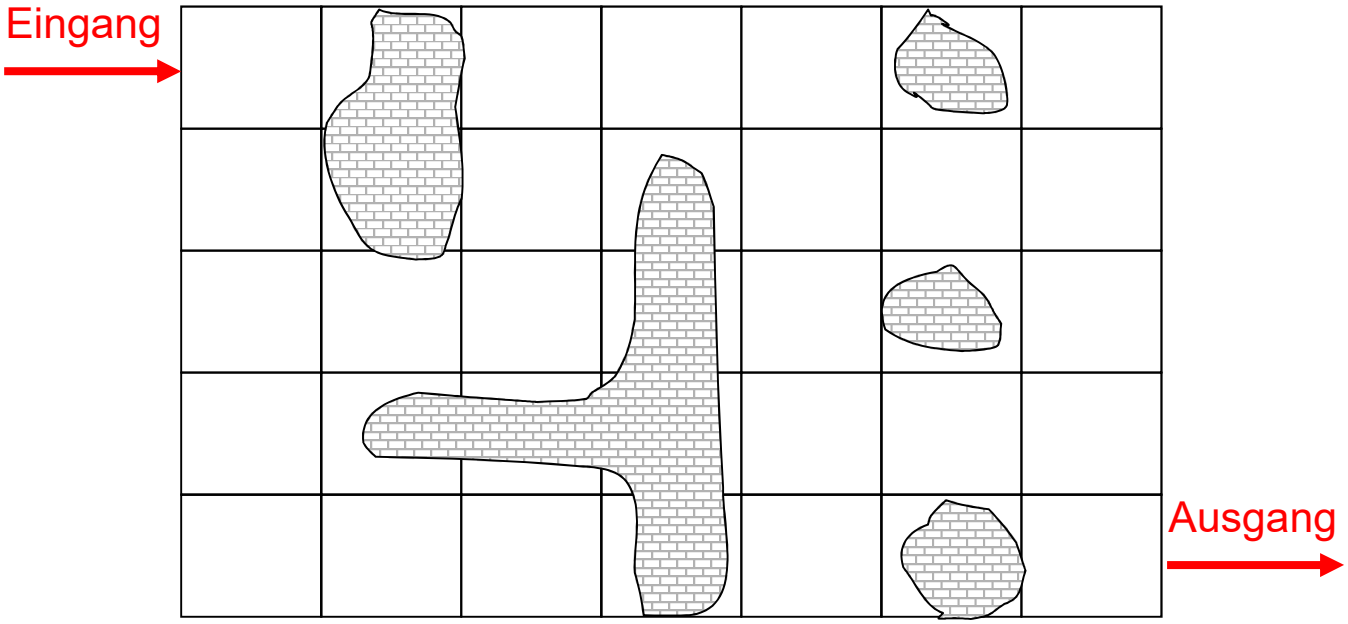


Modellbildung



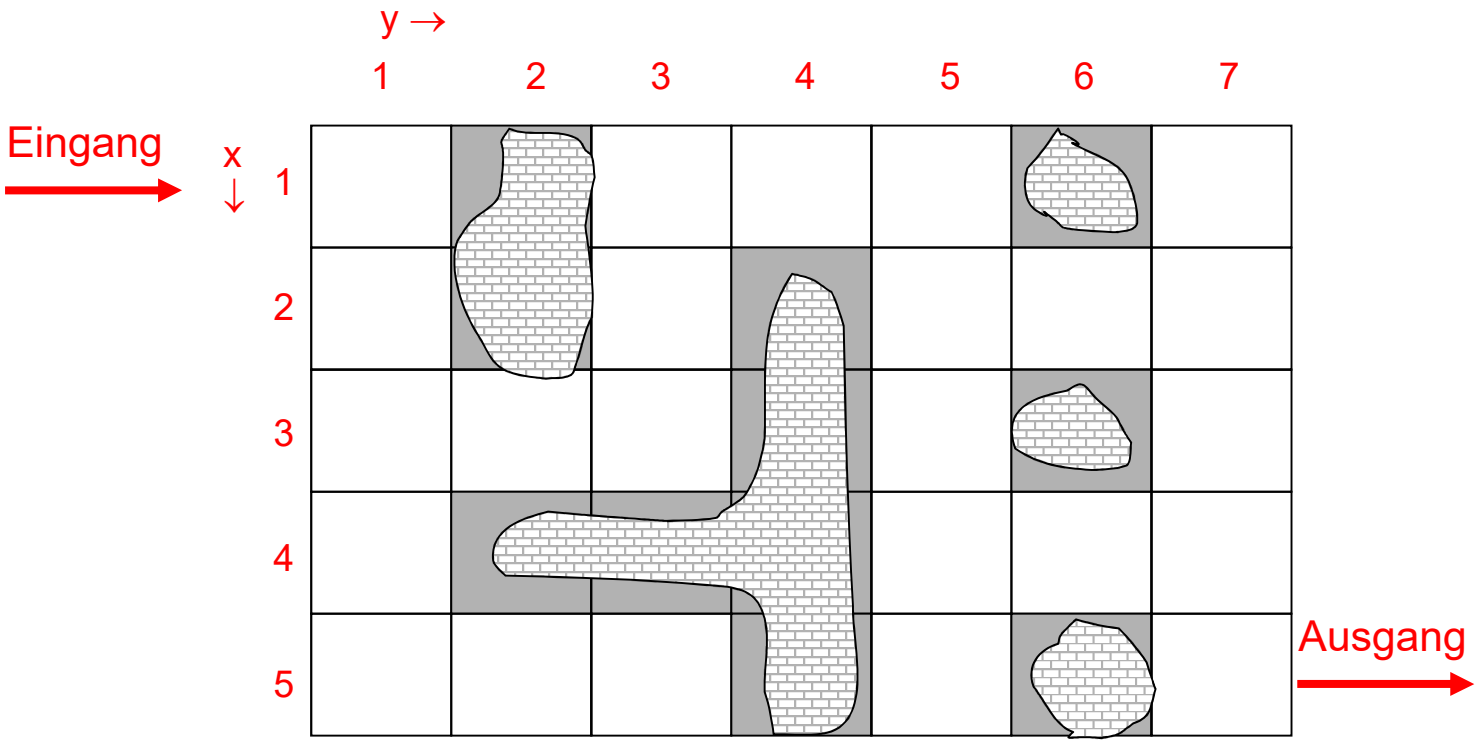
Finden eines geeigneten Modells

- Bilde die nicht-diskrete reale Labyrinthwelt auf ein diskretes Gitter ab



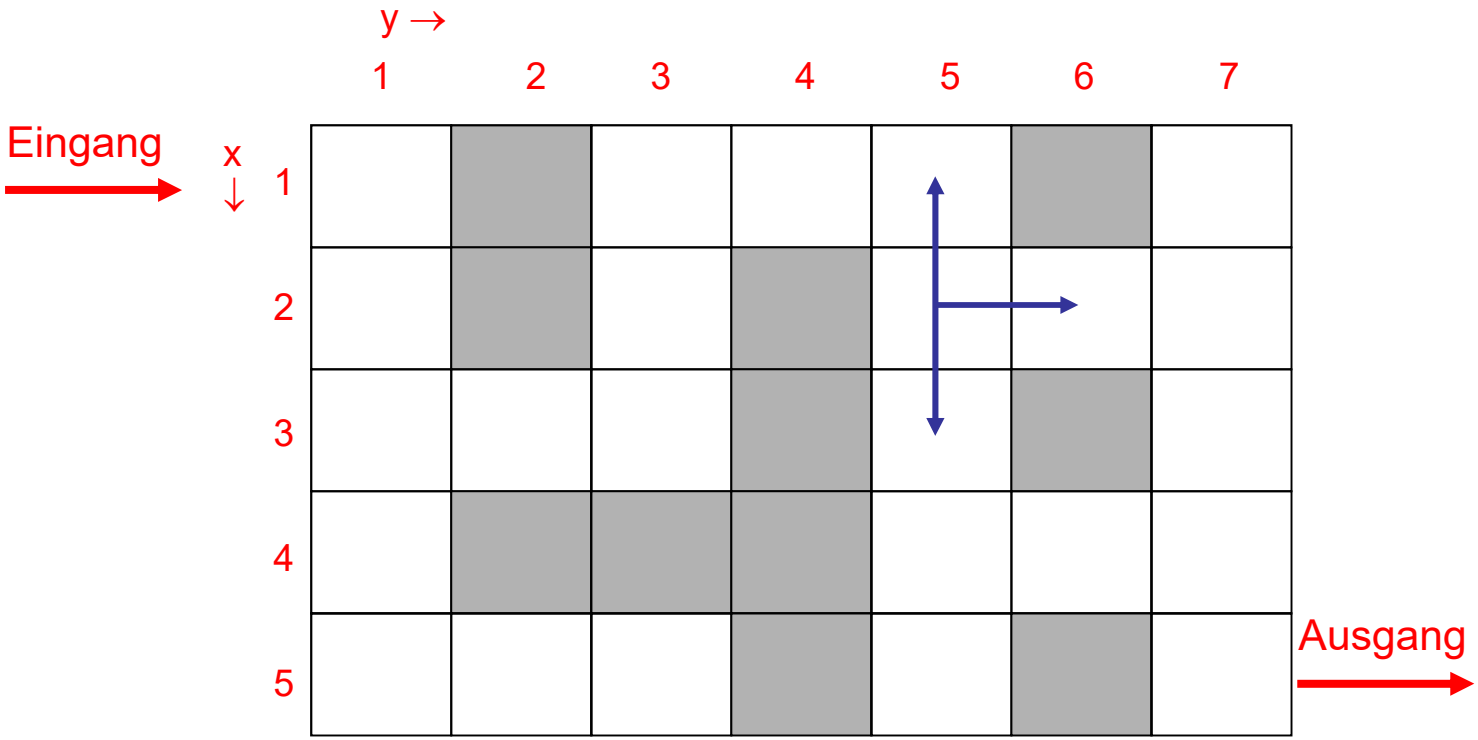
Labyrinthmodell

- In der Modellwelt ist ein Gitterrechteck entweder belegt (dort steht ein Hindernis) oder nicht belegt (kein Hindernis vorhanden)
- Rechtecke lassen sich eindeutig durch x- und y-Koordinate bezeichnen



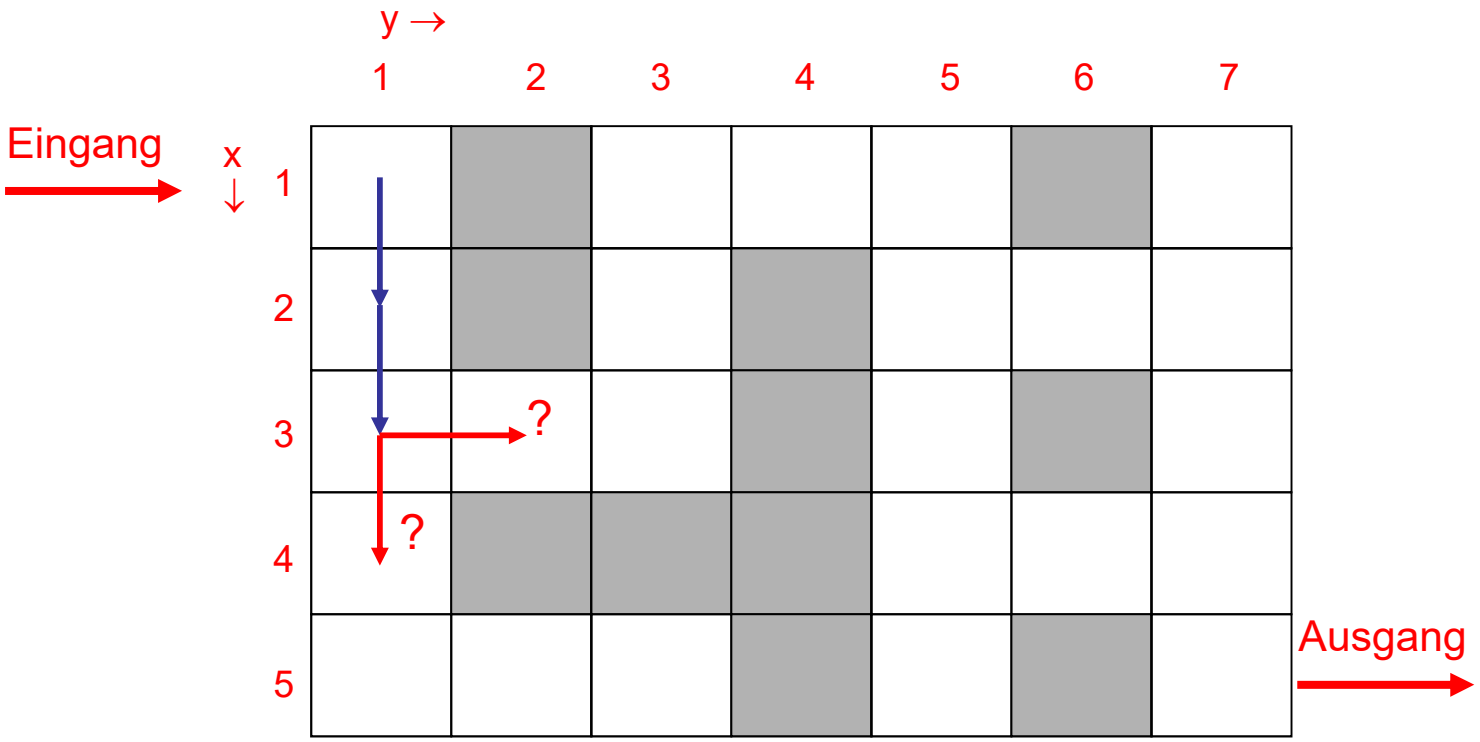
Labyrinthmodell

- Von einem Rechteck aus kann man sich nur zu einem freien Nachbarrechteck bewegen
- Von einem freien Rechteck (x,y) gibt es maximal 4 Möglichkeiten der Fortbewegung: (x-1,y), (x+1,y), (x,y-1), (x,y+1)
- Im Bild sind die möglichen Bewegungen von (2,5) aus eingezeichnet



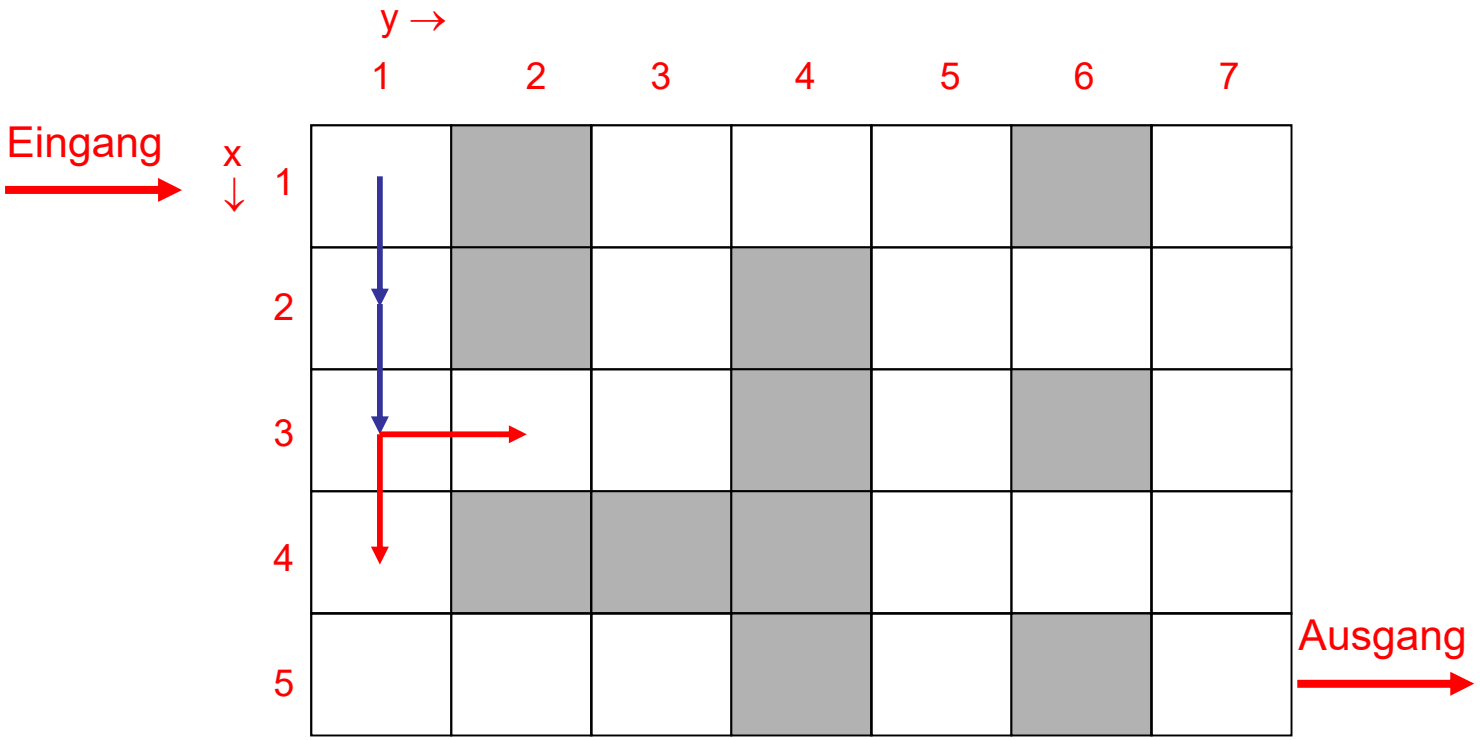
Entwicklung einer Strategie

- Beginne beim Startpunkt
- Einfach: Wo es keine Alternativen zum weiteren Vorgehen gibt, vollziehe den einzigen möglichen weiteren Schritt
- **???:** Falls es an einem Punkt (x,y) Alternativen zum weiteren Vorgehen gibt, was sollen wir tun?



Entwicklung einer Strategie

- Beginne beim Startpunkt
- Einfach: Wo es keine Alternativen zum weiteren Vorgehen gibt, vollziehe den einzigen möglichen weiteren Schritt
- **!!!:** Falls es an einem Punkt (x,y) Alternativen zum weiteren Vorgehen gibt, untersuche alle möglichen Wege, die von diesem Punkt aus starten!



Entwicklung einer Strategie

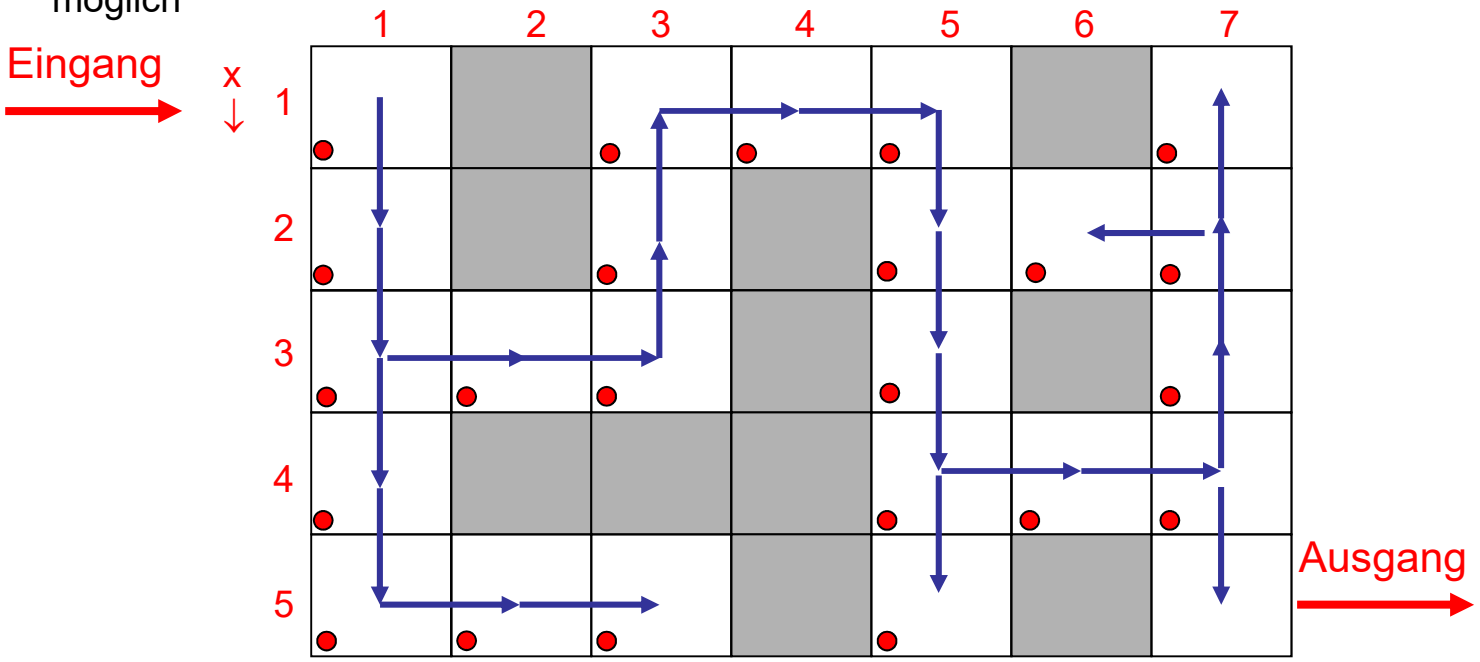
- Verfolge alle möglichen Wege
- Falls der Endpunkt eines Weges gleich dem Ausgang ist sind wir fertig
- Falls wir auf einem Weg an einem Punkt ankommen, an dem wir schon alle möglichen Fortsetzungen untersucht haben, brauchen wir an dieser Stelle nicht weiter zu suchen (wir kennen das Ergebnis ja schon)

Umsetzung der Strategie

- Beginne beim Eingang, d.h. am Punkt (1,1)
- An einem Punkt (x,y) wende folgende Strategie an:
  - Ist der Punkt (x,y) der Ausgang, sind wir fertig!
  - Falls der Punkt markiert ist, breche die weitere Suche ab (wir waren schon dort).
  - Ansonsten:
    - Markiere den Punkt (x,y) als besucht
    - Verfolge (falls möglich) den Weg, der von (x-1,y) ausgeht ("nach oben")
    - Verfolge (falls möglich) den Weg, der von (x+1,y) ausgeht ("nach unten")
    - Verfolge (falls möglich) den Weg, der von (x,y+1) ausgeht ("nach rechts")
    - Verfolge (falls möglich) den Weg, der von (x,y-1) ausgeht ("nach links")
    - Falls kein von (x,y) ausgehender Weg erfolgreich war, gib als Ergebnis an: "kein Weg möglich"

Beispiel

- Ist der Punkt (x,y) der Ausgang, sind wir fertig!
- Falls der Punkt markiert ist, breche die weitere Suche ab.
- Ansonsten:
  - Markiere den Punkt (x,y) als besucht
  - Verfolge (falls möglich) den Weg, der von (x-1,y) ausgeht ("nach oben")
  - Verfolge (falls möglich) den Weg, der von (x+1,y) ausgeht ("nach unten")
  - Verfolge (falls möglich) den Weg, der von (x,y+1) ausgeht ("nach rechts")
  - Verfolge (falls möglich) den Weg, der von (x,y-1) ausgeht ("nach links")
  - Falls kein von (x,y) ausgehender Weg erfolgreich war, gib als Ergebnis an: "kein Weg möglich"



Bemerkungen

- Der komplette Java-Code ist auf unserer Homepage
- Dort ist auch enthalten, wie man die besuchten Stationen auf dem Weg auf dem Bildschirm ausgeben kann

- Man kann durch eine leichte Modifikation des Algorithmus anstatt eines Weges auch alle möglichen Wege ermitteln, die zum Ziel führen

- Durch die Wahl der Gittergröße (Auflösung) können eventuell Lösungen (Wege) im Modell nicht gefunden werden, die für das reale Problem existieren!

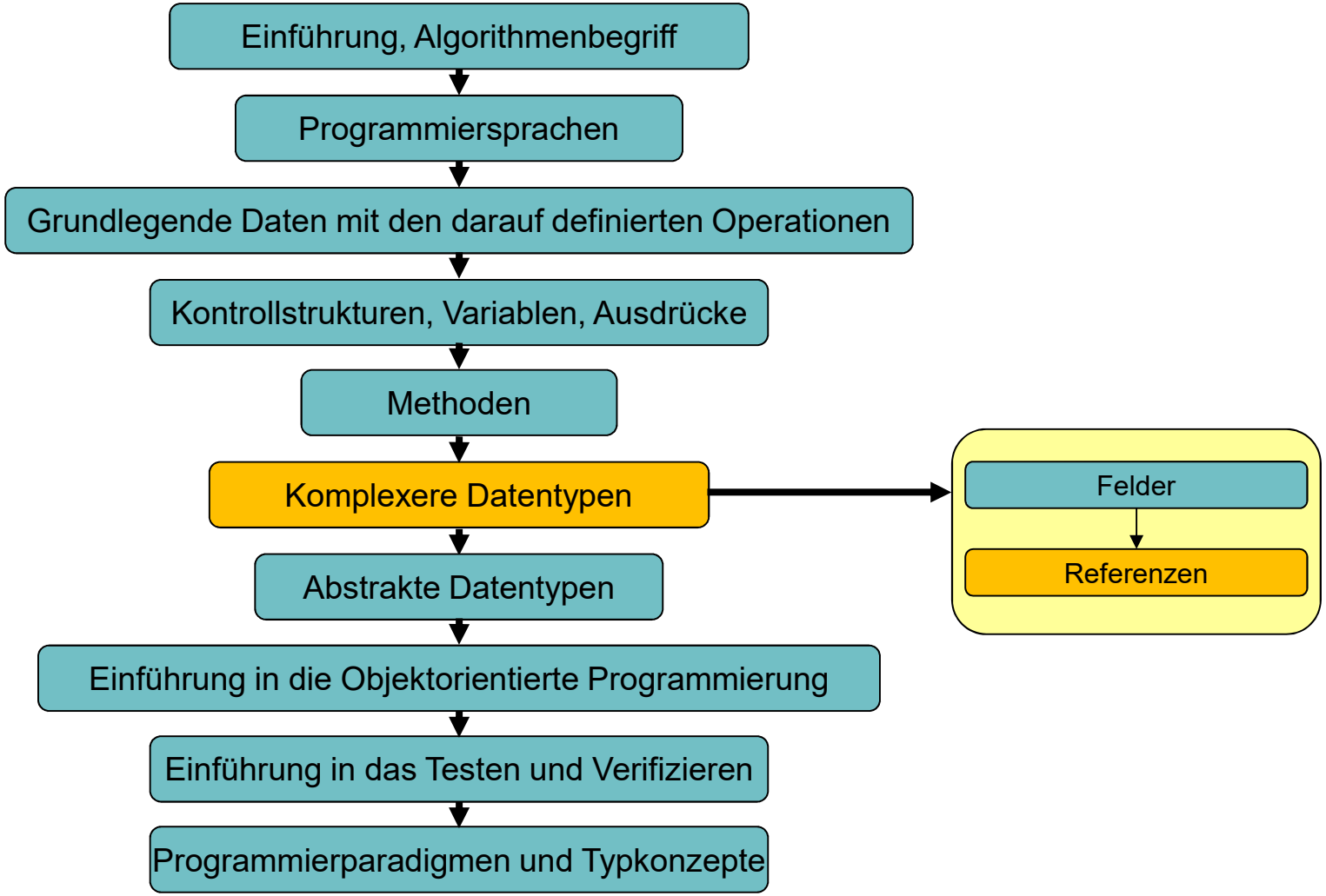
Zwischenstand

- Felder sind eine Anordnung von n Elementen eines Basistyps
- In Java ist `T[]` ein Feldtyp zu einem beliebigen Basistyp `T`
- In Java wird zwischen Feldvariablen und Feldobjekten unterschieden
- Die Länge eines Feldes `a` ermittelt man über `a.length`
- Feldobjekte werden mit `new` / implizit über den Feldinhalt angelegt
- Auf ein Feldelement greift man über einen ganzzahligen Index `i` zu: `a[i]`
- Mehrdimensionale Felder sind Felder von Feldern, die Elemente in der ersten Dimension sind also selber wieder Felder und werden so behandelt
- Backtracking ist eine Strategie, in der man an einem Entscheidungspunkt früher getroffene Entscheidungen zu diesem Punkt zurücknehmen kann und stattdessen eine weitere Alternative an dieser Stelle verfolgen kann

Reflektion

- Wäre das möglich: `(new int[]{1,2,3}).length` ? Wenn nein, wieso nicht? Wenn ja, was wäre das Ergebnis?

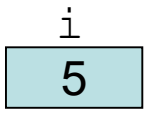
Inhalt dieser Veranstaltung



Referenzen

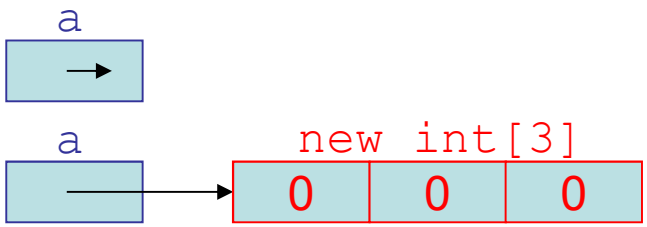
- Bis jetzt: Variablen eines primitiven Datentyps haben als Inhalt den aktuellen Wert der Variablen (ein ganzzahliger Wert, Fließkommawert,...)
- Beispiel:

```
int i = 5;
```



- Jetzt neu (und anders!): Variablen eines Feldtyps (allgemein Referenztyps) haben als Inhalt eine Referenz / Zeiger auf ein Feld und nicht das Feld selber
- Beispiel:

```
int[] a;

a = new int[3];
```



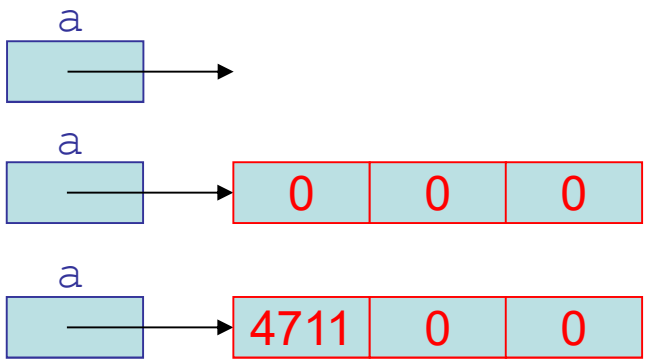
- Referenzen sind Hauptspeicheradressen
- Der Wert eines new-Ausdrucks ist die Speicheradresse (Referenz, Zeiger), ab der das neu erzeugte Feld im Speicher (genauer im Heap) zu finden ist. Diese Referenz wird durch eine Zuweisung in einer Variablen gespeichert.

Referenzvariablen

- Über eine Referenzvariable kann man sich auf das referenzierte Objekt beziehen / damit arbeiten
- Der Compiler führt automatisch die Derefenzierung durch

- Beispiel:

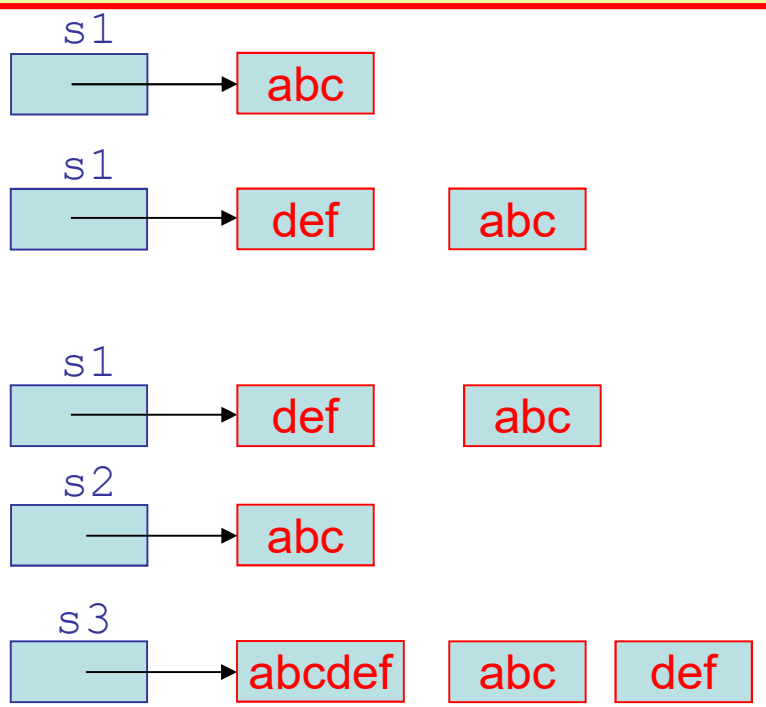
```
int[] a;

a = new int[3];

a[0] = a[0] + 4711;
```



Weitere Eigenschaften von Strings

- Variablen vom Typ `String` (auch ein Referenztyp!) enthalten Referenzen
- Jede String-Konstante ist ein Objekt im Hauptspeicher mit dem Stringinhalt, auf das eine Referenz in einer (oder mehreren) Variablen gespeichert werden kann
- Strings werden in Java nie verändert! Das Ergebnis einer Stringoperation mit String-Ergebnis ist immer ein neuer String.
- Beispiel:

```
String s1 = "abc";

s1 = "def";

String s2 = "abc";

String s3 = "abc"+"def";
```



Vergleich von Referenzen

- Wichtig: vergleicht man zwei Referenzen vom Typ `String` oder einem beliebigen anderen Referenztyp mit ==, so werden die **Referenzen** (Hauptspeicheradressen) verglichen und **nicht der Inhalt** der referenzierten Objekte!

- Will man den Inhalt von Strings vergleichen, muss man die String-Methode `equals` nehmen (es gibt eine weitere Alternative; siehe Dokumentation)

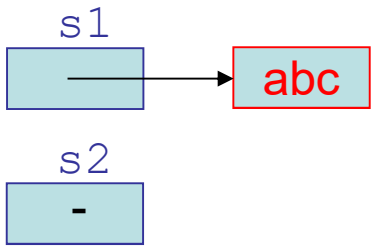
- Beispiel:

```
String s1 = "abc";
String s2 = "ab";
String s3 = „c";
String s4 = s2 + s3;
boolean b1 = (s1 == s4);          // false
boolean b2 = s1.equals(s4);       // true
```

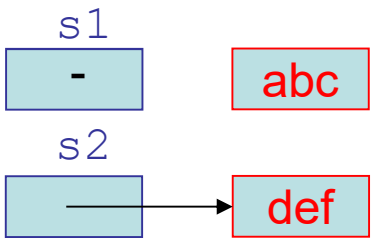
null

- Was ist, wenn eine Referenzvariable auf kein Objekt verweist oder nicht mehr verweisen soll?
- Dazu gibt es den Wert `null`, der verschieden ist von allen möglichen Referenzen
- Beispiel:

```
String s1 = "abc";
String s2 = null;
```




```
s1 = null;
s2 = "def";
```

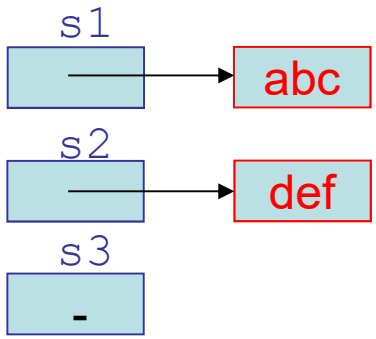



```
if((s1 == null) || (s2 == null))
        ...
```

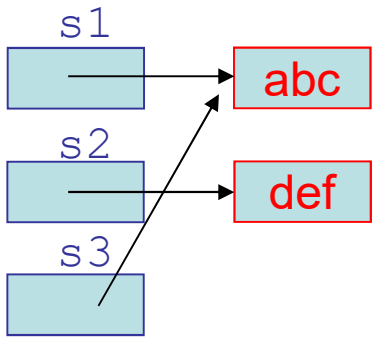
Referenzen sind Werte

- Eine Referenz ist ein Wert eines Referenztyps, den man z.B. kopieren kann
- Regeln zur Typanpassung bei Referenzen werden hier nicht behandelt (später)
- Beispiel:

```
String s1 = "abc";
String s2 = "def";
String s3 = null;
```




```
s3 = s1;
```



- **Sehr wichtig**: Damit ist es möglich, dass auf ein Objekt mehrere Referenzen zeigen können. Manchmal ist das sehr nützlich, manchmal entstehen dadurch Probleme.

swap wiederbesucht

Alte Version mit 2 int-Variablen:

```
public class Uebergabestrategien {
 public static void main(String[] args){
  int a=2, b=3;
  swap(a,b);
  System.out.println("a="+a+", b="+b);
 }

 public static void swap(int x, int y) {
  int tmp;
  tmp = x;
  x = y;
  y = tmp;
  return;
 }
}
```

Ausgabe:

```
a=2, b=3
```

Neue Version mit Feld-Variable:

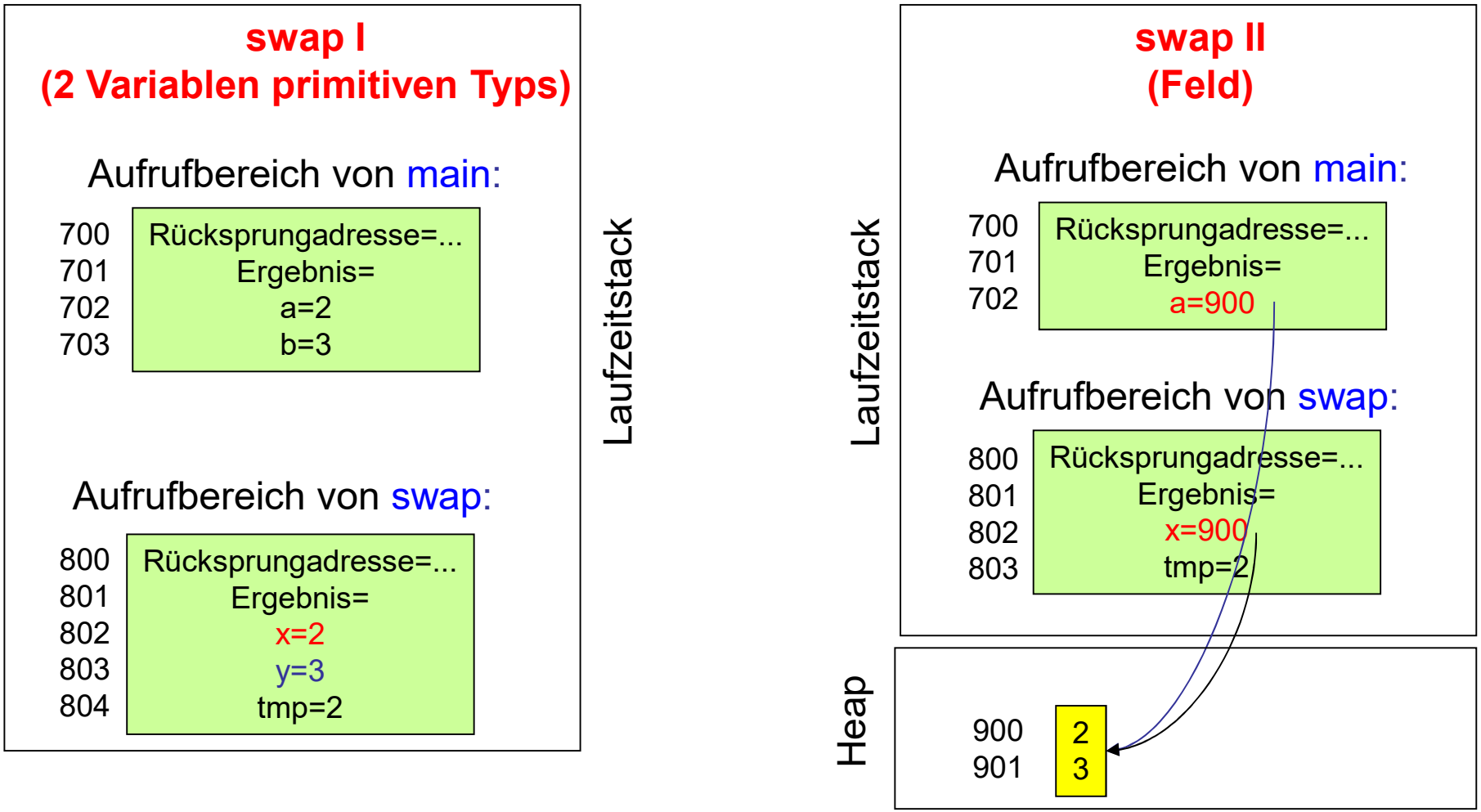
```
public class Uebergabestrategien2 {
 public static void main(String[] args){
  int[] a = {2, 3};
  swap(a);
  System.out...("a="+a[0]+", b="+a[1]);
 }

 public static void swap(int[] x) {
  int tmp;
  tmp = x[0];
  x[0] = x[1];
  x[1] = tmp;
  return;
 }
}
```

Ausgabe:

```
a=3, b=2
```

Grund: über Referenz greift man in der Methode auf Originalwerte zu

Aufrufbereiche bei swap I+II



Zwischenstand

- Neben primitiven Typen gibt es Referenztypen, die komplett anders behandelt werden
- Referenzen sind Hauptspeicheradressen
- Eine Variable eines Referenztyps enthält immer eine Referenz
- `null` ist ein spezieller Referenzwert
- Auf ein Objekt können 0, 1 oder mehrere Objekte zeigen

Reflektion

- Ergänzen Sie den Programmausschnitt so, dass verschiedene Werte in der Programmausgabe erscheinen, ohne dass die Variable `a` auf der linken Seite einer Zuweisung erscheint:

```
int[] a = {1}; System.out.println(a[0]); System.out.println(a[0]);
```

Zusammenfassung

- In Java sind zu einem beliebigen Basistyp `T` Feldtypen beliebiger Dimension möglich
- Ein konkretes Feld zu einem Feldtyp muss mit dem new-Operator angelegt werden (bzw. über einen Initialisierungsausdruck)
- Der Zugriff auf die Feldelemente geschieht in einer Form wie `a[i]`

- Feldtypen sind Referenztypen (wie auch der Typ `String`)
- Variablen eines Referenztyps enthalten immer nur eine Referenz auf das eigentliche Feld / Objekt
- Es kann auf ein Feld keine, eine oder mehrere Referenzen in Variablen geben
- Eine Variable eines Referenztyps kann `null` enthalten oder eine Referenz auf ein Objekt dieses Referenztyps
- Der Vergleich zweier Referenzen (aber nicht der Inhalte) mit `==` ist möglich
- In einer Methode hat man bei einem Referenzwert als Übergabeparameter Zugriff auf den Originalwert des Objektes