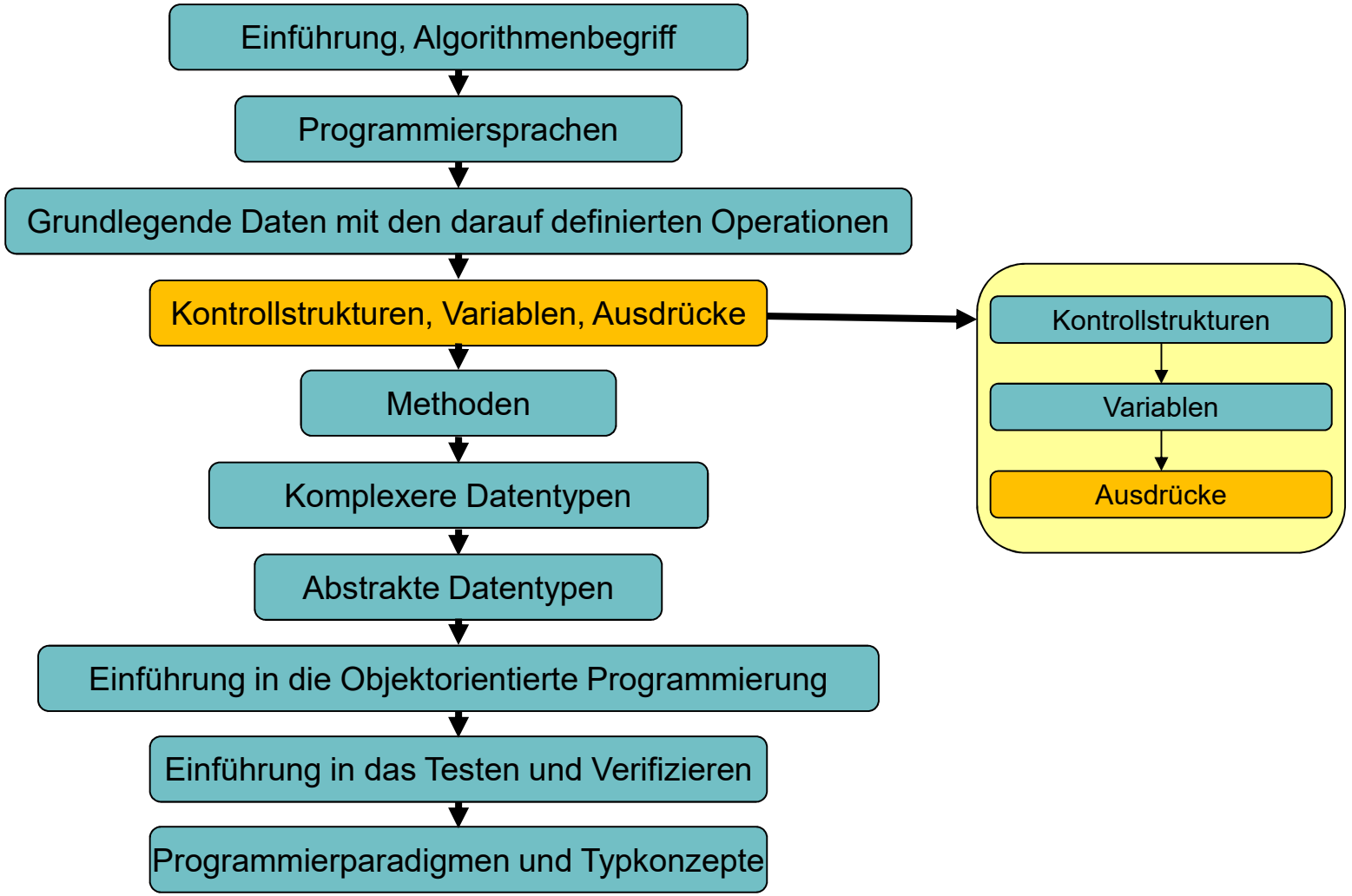
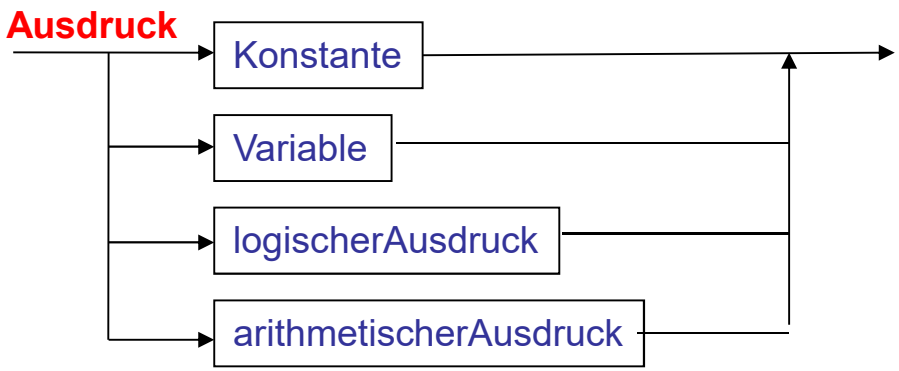
Inhalt dieser Veranstaltung



Ausdruck

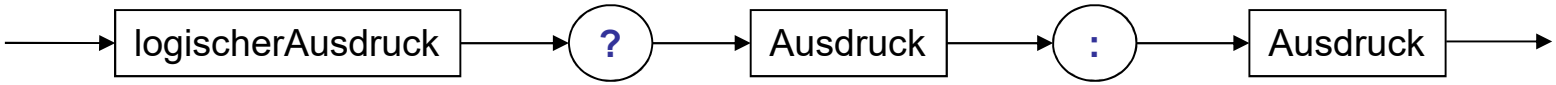
- Ein Ausdruck gibt eine Berechnungsformel an
- Ein Ausdruck kann ausgewertet werden und liefert dann genau einen Wert
- Ein Ausdruck (und der berechnete Wert) hat einen bestimmten Typ, der sich aus den darin vorkommenden Operatoren und Operanden ergibt
- Ein logischer Ausdruck ist vom Typ `boolean`, der Wert eines arithmetischen Ausdrucks ist von einem numerischen Typ

- Bis jetzt bekannt:



Bedingungsausdruck

Bedingungsausdruck



- Bedeutung:
  - Der logische Ausdruck wird ausgewertet.
  - Falls dieser Wert `true` ist, so ist das Ergebnis des Gesamtausdrucks der Wert des Ausdrucks nach dem Fragezeichen, ansonsten der Wert des Ausdrucks nach dem Doppelpunkt

- Von den beiden Ausdrücken wird nur der ausgewertet, der sich aufgrund des logischen Ausdrucks ergibt (wichtig bei Seiteneffekten)

- Dies entspricht einer if-else-Anweisung in Ausdrucksform und kann damit in einem umfassenden Ausdruck genutzt werden

Beispiel Fehlerabschätzung

alte Version mit if-else

```
do {
    // Berechnungsvorschrift
    y = 0.5 * (y + x/y);
    // Absolutbetrag fuer Fehlerabschaetzung
    if(x > y*y)
        fehler = x - y*y;
    else
        fehler = y*y - x;
} while(fehler > epsilon);
```

Nutzung Bedingungsoperator

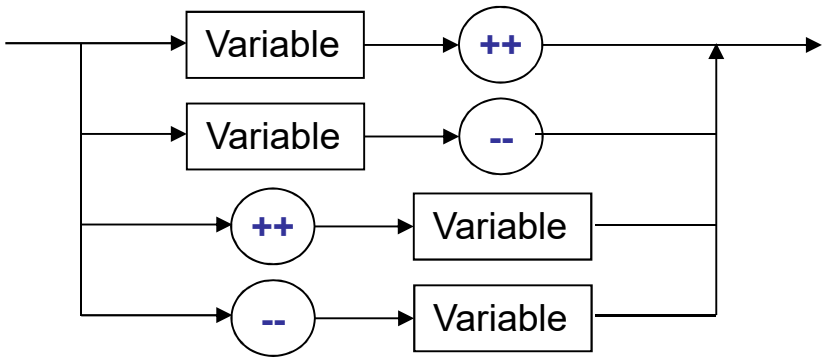
```
do {
    // Berechnungsvorschrift
    y = 0.5 * (y + x/y);
    // Absolutbetrag fuer Fehlerabschaetzung
    fehler = (x > y*y) ? (x - y*y) : (y*y - x);
} while(fehler > epsilon);
```

Nutzung Bedingungsoperator in weiterem Ausdruck
(allerdings nicht gut lesbar)

```
do {
    // Berechnungsvorschrift
    y = 0.5 * (y + x/y);
} while((x > y*y) ? (x - y*y) : (y*y - x) > epsilon);
```

Inkrement-/Dekrementoperatoren

Inkrement-/Dekrementausdruck



- Häufig vorkommende Operation: erhöhe/erniedrige den Wert einer (ganzzahligen) Variablen um 1
- Beispiel: Update-Teil in Zählschleifen
- Diese Operation lässt sich auch in Form eines Ausdrucks verwenden
- Die Variable hat auf jeden Fall einen geänderten Wert (alter Wert + 1)
- Unterschied bei den Varianten:
  - Der Wert des Ausdrucks ist der alte Wert der Variablen: `a++`
  - Der Wert des Ausdrucks ist der neue Wert der Variablen: `++a`
- Häufiger Fehler: `a = a++;`

Beispiel

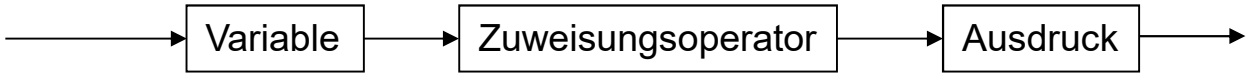
```
/**
 * Quadratzahlen und Kubikzahlen zwischen 10 und 20 erzeugen
 */
public class Quadratzahlen2 {
    public static void main(String[] args) {
        // vorwaerts zaehlen
        for(int i=10; i <= 20; i++) {
            System.out.println(i + " " + (i*i) + " " + (i*i*i));
        }

        // rueckwaerts zaehlen
        for(int i=20; i >= 10; i--) {
            System.out.println(i + " " + (i*i) + " " + (i*i*i));
        }
    }
}
```

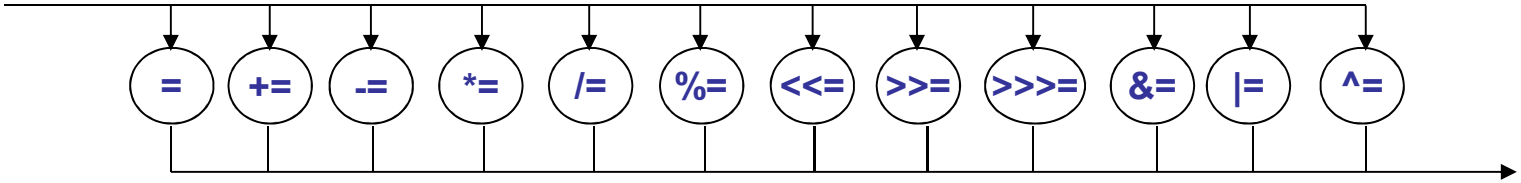
In den beiden hier gezeigten Beispielen ist es unerheblich, ob man `i++ / i--` oder `++i / --i` schreibt. Wieso?

Zuweisungsausdruck

Zuweisungsausdruck



Zuweisungsoperator



- Die einfachste Form ist die bereits bekannte Form der Zuweisung

```
variable = ausdruck
```

- Die anderen Formen entsprechen sinngemäß

```
variable = variable operator ausdruck
```

- Beispiel: `x +=2` entspricht `x = x + 2`

- Der Wert solch einer Zuweisung (ein Ausdruck!) ist der zugewiesene Wert
- Beispiel: `x = (y += 2);`

Beispiel

```
/**
 * ermittle die erste natuerliche Zahl groesser als 1000, die von 1
 * beginnend sich jeweils durch Verdopplung der Vorgaengerzahl ergibt
 */
public class LeereAnweisung {
    public static void main(String[] args) {

      int zahl;

      for(zahl=1; zahl<=1000; zahl+=zahl)
          // hier muss nichts mehr getan werden
          ;

      System.out.println("Die gesuchte Zahl ist " + zahl);
    }
}
```

Zwischenstand

- Jeder Ausdruck liefert bei seiner Berechnung einen Wert
- Jeder Ausdruck hat einen Typ, der sich aus den Operatoren und Operanden ergibt
- Der Bedingungsoperator ist ein if-then-else-Konstrukt in Ausdrucksform
- Die zwei Formen der Inkrement-/Dekrement-Operatoren wirken unterschiedlich
- Zuweisungsoperatoren verändern den Wert einer Variablen (Update-Operation)

Reflektion

- Welche Werte haben die Variablen `x` und `y` nach folgender Anweisungsfolge und wieso?

```
int x = 5, y=5;
x = ++x + y++;
```

Überladen von Operatoren

- Manche Operatoren wurden bereits in mehreren Zusammenhängen verwandt

- Beispiel: +
  - `+3` : Vorzeichenplus
  - `3+4` : Addition der beiden int-Zahlen 3 und 4 in Zweierkomplementdarstellung
  - `3.0 + 4.0` : Addition der beiden Fließkommazahlen 3.0 und 4.0 in IEEE-Darstellung
  - `"abc" + "def"` : Verkettung von Strings

- Die konkrete Bedeutung des Operators ergibt sich aus dem Zusammenhang (Anzahl und/oder Typ der Operanden)

- Solche Operatoren nennt man überladen

Priorität von Operatoren

- Aus der Schulmathematik bekannt: Punkt-vor-Strichrechnung
- Beispiel: `3 + 4 * 5` bedeutet `3 + (4 * 5)`
- Multiplikationsoperatoren haben dabei eine höhere Priorität / eine stärkere Bindung als Additionsoperatoren

- In Programmiersprachen ist die Priorität **aller** Operatoren genau definiert
- Kommen in einem Ausdruck mehrere Operatoren vor, so werden diese in der Reihenfolge ihrer Priorität angewandt
- Beispiel: `3 + 4 * 5` wird ausgewertet als `3 + (4 * 5)`

- Bevor ein Operator angewandt wird, müssen alle Operanden ausgewertet werden (gleich: wenige Ausnahmen)

- Durch diese genaue Definition der Auswertungsreihenfolge im Java-Sprachstandard soll eine Reproduzierbarkeit von Ergebnissen auf allen Rechnern / mit allen Java-Compilern erreicht werden

Assoziativität von Operatoren

- Was passiert, wenn Operatoren mit gleicher Priorität vorhanden sind?
- Dann definiert die Assoziativität von Operatoren die Reihenfolge der Auswertung

- Die meisten Operatoren sind linksassoziativ, d.h. von links nach rechts wird in solchen Fällen ausgewertet

- Beispiel 1: `3 + 4 + 5` wird ausgewertet als `(3 + 4) + 5` , weil der Additionsoperator linksassoziativ ist
- Beispiel 2: `x = y = 5` wird ausgewertet als `x = (y = 5)` , weil der Zuweisungsoperator rechtsassoziativ ist

Übersicht über Priorität und Assoziativität

Operator	Beschreibung	Assoziativität
`()`	Klammerung	links
`[]`	Feldzugriff	links
`()`	Methodenaufruf	links
`.`	Zugriff auf Komponente	links
`++, --`	Inkrement/Dekrement	rechts
einstellige Operatoren wie `+, ~, !`		rechts
`*,/,%`	Multiplikationsoperatoren	links
`+,-`	Additionsoperatoren	links
`+`	Stringverkettung	links
`<<,>>,>>>`	Shift	links
`<,<=,>=,>`	Vergleich	links
`==, !=`	vergleich	links
`&,^,|`	bitweises/logisches Und/Xor/Oder	links
`&&, ||`	bitweises/logisches Und/Xor/Oder	links
`?:`	Bedingungsoperator	rechts
`=, +=` usw.	Zuweisungsoperator	rechts

- Einfache Merkregel: Klammern haben höchste Priorität
- Operatoren sind hier in Prioritätsklassen zusammengefasst (haben untereinander aber weitere Prioritätsgliederung)

Wieso &,&& und |,|| ?

- Die Operatoren `&` und `&&` arbeiten auf den ersten Blick gleich (Und-Operation)

- `x & y` ist die Und-Verknüpfung von `x` und `y`
- Arbeitsweise `&`: werte `x` aus, dann werte `y` aus, dann wende Und-Operator an
- Beim logischen Und-Operator gilt: ist ein Operand `false`, so kann das Gesamtergebnis nur `false` sein (unabhängig von zweiten Operanden!)
- Analog bei Oder
- Der Operator `&` erzwingt aber die Auswertung beider Operanden

- Alternativ kann man `&&` verwenden (analog `||`):
  - Werte den linken Operanden aus. Ist dieser `false`, so ist das Gesamtergebnis `false` (werte also den rechten Operanden **nicht** mehr aus!)
  - Ist der Wert des linken Operanden `true`, so werte den rechten Operanden aus und bilde mit den beiden Werten das Gesamtergebnis
- Diese Eigenschaft darf man im Programm ausnutzen

Unterschiede &,| und &&, ||

Fall 1: Seiteneffekt

```
int x = 0;
if( (x >= 0) | (--x < 0) ) {
    ...
}
```

1. `x >= 0` wird ausgewertet und liefert den Wert `true`
2. `--x < 0` wird ausgewertet. Dazu wird `x` dekrementiert, hat also anschließend den Wert -1. Dann wird -1 mit 0 verglichen, was `true` ergibt
3. `true` ∨ `true` ergibt `true` und x hat den Wert -1

```
int x = 0;
if( (x >= 0) || (--x < 0) ) {
    ...
}
```

1. `x >= 0` wird ausgewertet und liefert den Wert `true`
2. Das Gesamtergebnis ist damit `true` und x hat den (alten) Wert 0

Fall 2: nicht-abbrechende Berechnung

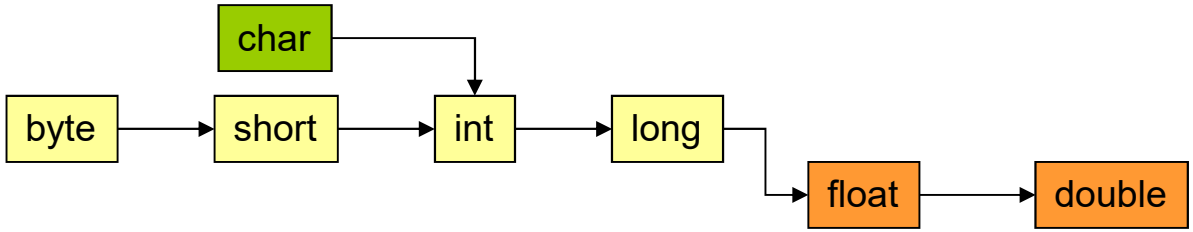
```
int x = 0;
if( (x >= 0) | f(x) ) {
    ...
}
```

Falls die Berechnung von f(x) nicht abbricht, so ist das Ergebnis des Gesamtausdrucks nicht definiert (man kommt nie zu einem Ergebnis).
Bei `||` ist das Ergebnis sehr wohl definiert.

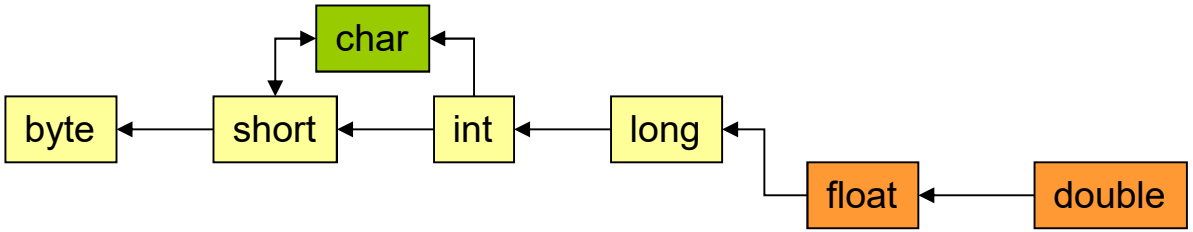
Typumwandlungen

- Was passiert, wenn ich `3 + 4.0f` schreibe?
  - Die Konstante `3` ist eine int-Konstante mit interner Zweierkomplementdarstellung mit 32 Bits
  - Die Konstante `4.0f` ist eine float-Konstante mit einer internen IEEE-Darstellung mit 32 Bits
  - Weder die int-Addition noch die float-Addition ist anwendbar!
- Deshalb gewünscht und notwendig: implizite und explizite Typumwandlungen
- Prinzipiell zu berücksichtigende Fälle:
  - `3 + 4l` ist problemlos als long-Addition `3l + 4l` durchführbar, da alle int-Werte exakt zu einem long-Wert umgewandelt werden können
  - Der double-Wert `3.0` lässt sich exakt zu einem int-Wert `3` umwandeln
  - Der double-Wert `3.5` lässt sich nicht exakt zu einem ganzzahligen Wert umwandeln
  - Die Umwandlung eines double-Wertes zu einem Wahrheitswert macht keinen Sinn
- In Java lassen sich Werte aller numerischen Typen (alle primitiven Typen außer `boolean`) prinzipiell in einen anderen numerischen Typ umwandeln

Erweiternde und einengende Typumwandlung



- **Erweiternde** Typumwandlungen können implizit stattfinden (gleich mehr)
- Beispiel: 3 + 4.0
- Innerhalb der ganzzahligen Typen sind diese Umwandlungen exakt
- Der Übergang von ganzzahlig nach Fließkomma muss nicht exakt sein



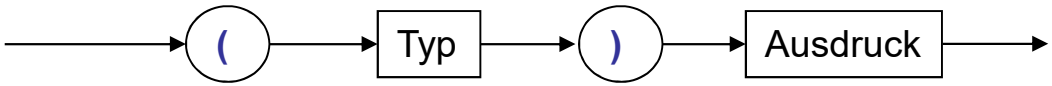
- **Einengende** Typumwandlungen finden nur explizit statt (cast-Ausdruck)
- Beispiel: Umwandlung von `4.5` nach `int`: `(int)4.5`
- Die Umwandlungen müssen nicht exakt sein
- Bei der Umwandlung innerhalb ganzzahliger Typen werden die obersten n Bits abgeschnitten. Das kann sogar zu einer Vorzeichenumkehr führen!

Umwandlungsregeln

- Implizite Umwandlung in Form einer erweiternden Umwandlung finden statt:
  - in Zuweisungen. Beispiel `float x = 3;`
  - wenn in arithmetischen Ausdrücken die Typen nicht übereinstimmen. Beispiel: `3 + 4.0`
  - später: aktuelle Argumente von Methodenaufrufen auf Parametertyp
  - Sind Operanden in einem arithmetischen Ausdruck vom Typ `byte`, `short` oder `char`, so werden diese Werte immer zuerst auf `int` erweitert

- Explizite Umwandlungen (erweiternd oder einengend) nur in Form eines cast-Ausdrucks möglich
- Dabei gibt man in Klammern vor einem Ausdruck den Wunsch-Typ an
- Beispiel: `int y = (int)4.5;`
- Bei der Umwandlung von einem Fließkommatyp auf einen ganzzahligen Typ wird nur der ganzzahlige Anteil genommen (abgeschnitten)

castAusdruck



Zwischenstand

- Operatoren können überladen sein (in mehreren Zusammenhängen, evtl. sogar in unterschiedlicher Bedeutung nutzbar)
- Alle Operatoren haben eine definierte Priorität und Assoziativität
- | / || und & / && zeigen Unterschiede in der Auswertung des 2. Operanden
- Mit Typumwandlungen lassen sich Werte in einen anderen Typ umwandeln. Dies ist prinzipbedingt nicht immer möglich.
- Manche Umwandlungen finden implizit statt, andere nur explizit durch eine cast-Operation.

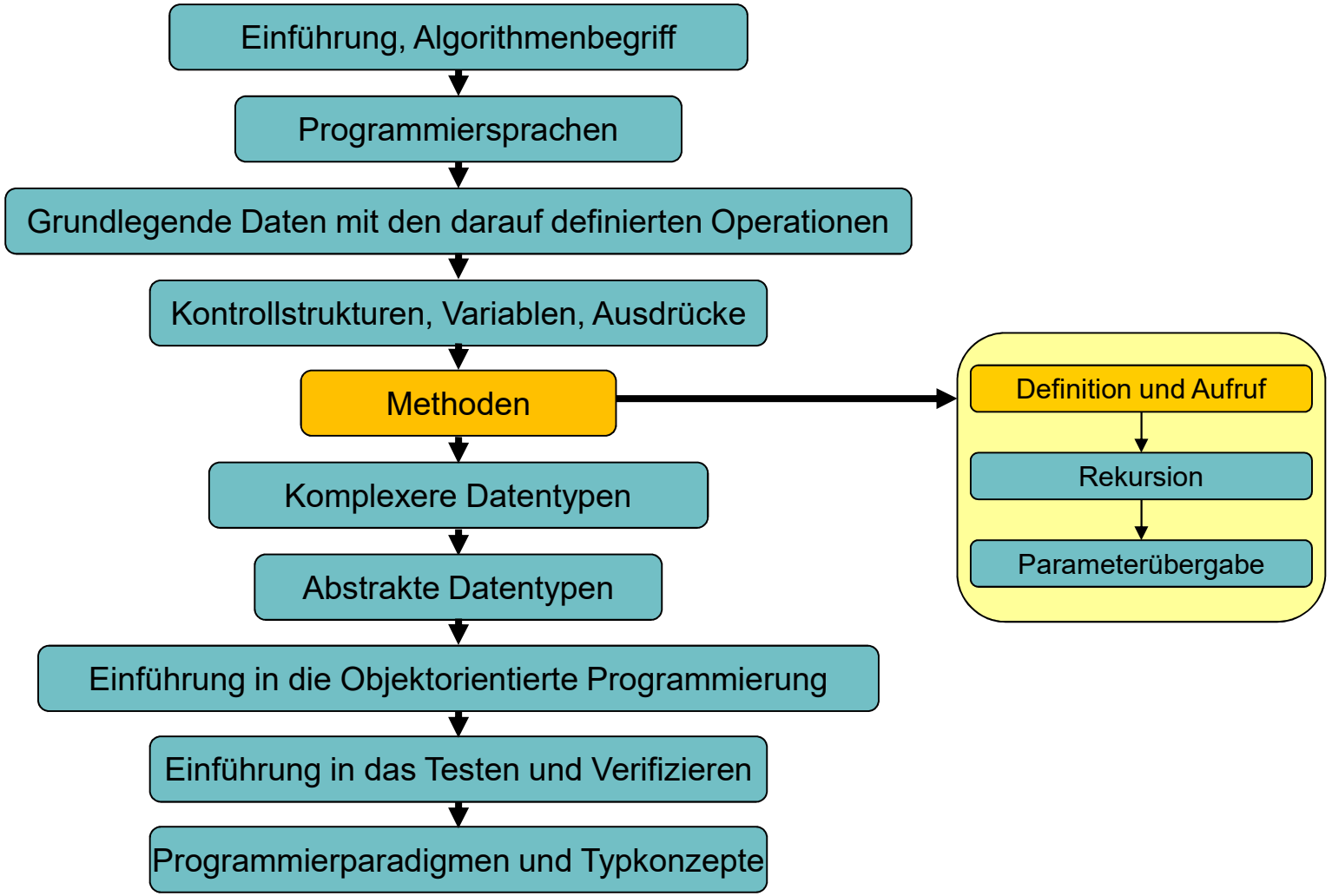
Reflektion

- Geben Sie ein Beispiel an, wo aus einem positiven Wert durch eine cast-Operation ein negativer Wert entsteht.

Zusammenfassung

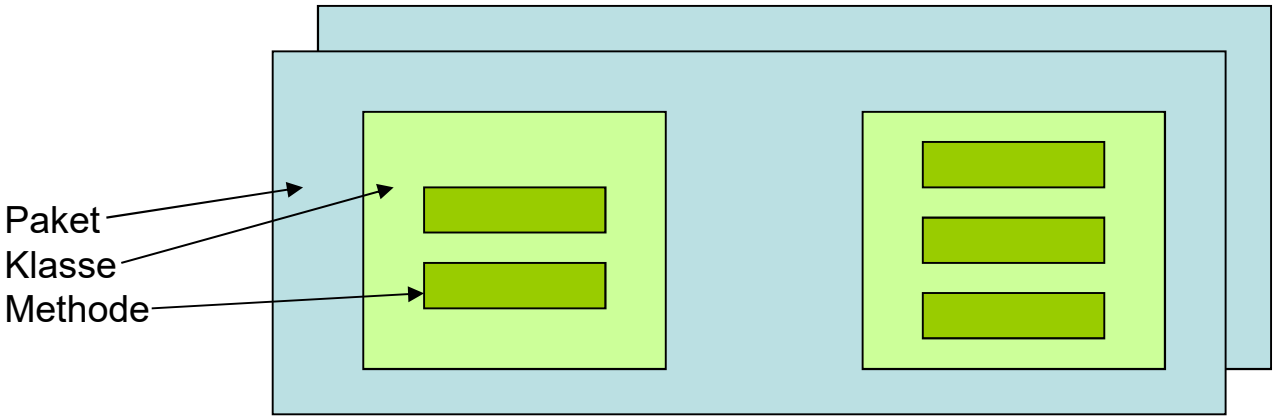
- Ein Ausdruck ist eine Berechnungsformel, die bei Auswertung genau einen Wert liefert
- Ausdrücke sind typisiert
- Operatoren können überladen sein
- Durch genau Prioritäts- und Assoziativitätsregeln (in Java) will man auf allen Rechnern eine Reproduzierbarkeit von Ergebnissen erreichen
- Typumwandlungen in Form von erweiternden und einengenden Umwandlungen
- Implizite Umwandlungen nur erweiternd möglich, explizite Umwandlungen in Form eines cast-Ausdrucks sowohl erweiternd als auch einengend möglich

Inhalt dieser Veranstaltung



Strukturierung großer Programme

- Bisherige Beispielprogramme sind wenige Zeilen lang
- Was ist bei großen Programmen (Mio. Programmzeilen)? Alles in `main()`?
- Antwort natürlich: nein
- Java-Programme können und sollten auf mehreren Ebenen strukturiert werden:
  - Blöcke: kennen wir schon
  - Methoden (Funktionen, Prozeduren, Unterprogramme,...): jetzt
  - Klassen: später
  - Pakete: später



- Diese Strukturierungselemente unterstützen
  - den systematischen Aufbau großer Programme
  - die Kapselung von Funktionalität und Daten
  - die Wiederverwendung von Programmcode

Definition und Auswertung einer Funktion

- Beispiel: Kennt man die Länge und Breite eines beliebigen Rechtecks, so lässt sich daraus die Fläche berechnen
- In der Mathematik kann man diesen allgemeinen funktionalen Zusammenhang durch die **Definition** einer für alle Rechtecke gültigen Funktion ausdrücken:

  `flaeche : R × R → R`

  `flaeche(Länge,Breite) := Länge · Breite`

- Will man konkret die Fläche eines Rechtecks berechnen, so kann man diese Funktion mit 2 konkreten Werten für Länge bzw. Breite **auswerten**

  `flaeche(5,20)` ergibt bei Auswertung den Wert `100`

- Auch die Fläche eines weiteren Rechtecks lässt sich einfach berechen:

  `flaeche(12,2)` ergibt bei Auswertung den Wert `24`

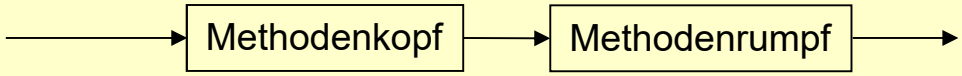
- Programmiersprachen setzen diese Idee ebenfalls um
- In Java realisiert man diesen Funktionsansatz der Mathematik durch Methoden

1) Definition einer Methode

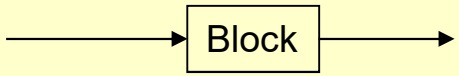
Notation in der Mathematik: flaeche : $R \times R \rightarrow R$ mit flaeche(x,y) := $x \cdot y$

Notation in Java:

Methodendefinition

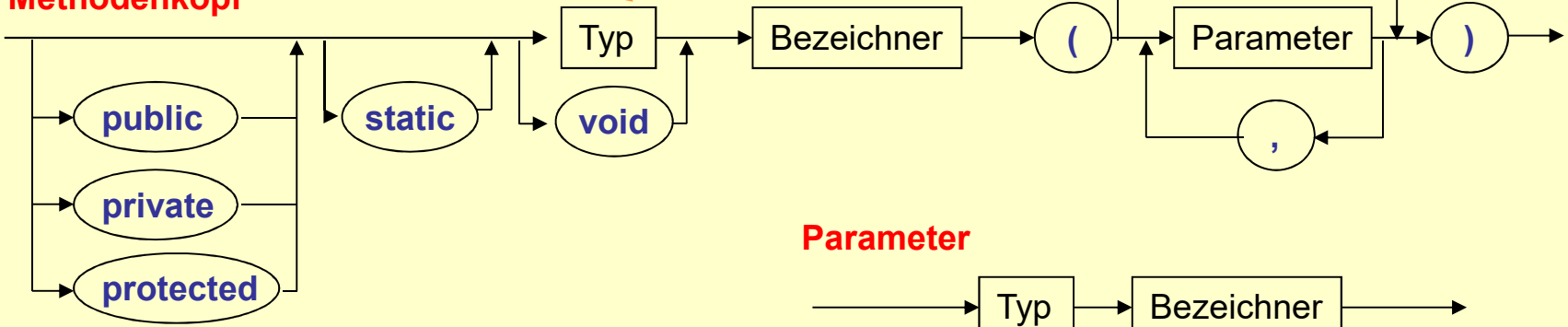


Methodenrumpf



flaeche : $R \times R \rightarrow R$

Methodenkopf



Parameter

Beispiel

```
/**
 * Flaechenberechnung von Rechtecken
 */
public class Flaechenberechnung {
    public static void main(String[] args) {
        ...
    }

    // Hier steht der Algorithmus zur Berechnung der Flaeche
    public static double flaeche(double laenge, double breite) {
        return laenge * breite;
    }
}
```

Methodenkopf

Methodenrumpf

Ergebnistyp

Methodenname

formale Parameter

- Methoden werden auf der gleichen Ebene wie main() angegeben.
- Der Gültigkeitsbereich des Methodennamens ist die gesamte Klasse
- Bis auf Weiteres schreiben wir immer `public static` vor eine Methode

- Der Methodenkopf beschreibt die Schnittstelle der Methode
- Der Methodenrumpf gibt die Implementierung der Methode an.
- Der Methodenrumpf ist formal ein Block. In diesem Block steht der Algorithmus zur Berechnung des Methoden-/Funktionswertes.

Formale Parameter

```
public class Flaechenberechnung {
    ...

    // Hier steht der Algorithmus zur Berechnung der Flaeche
    public static double flaeche(double laenge, double breite) {
        return laenge * breite;
    }
}
```

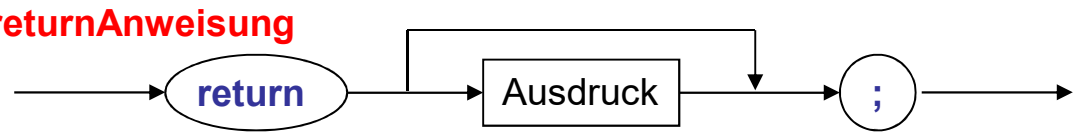
- Die formalen Parameter sind Platzhalter für Werte, die in einem Methoden**aufruf** an die Methode übergeben werden
- Ihr Name ist frei wählbar
- Die formalen Parameter werden wie Variablen des Rumpfs betrachtet (damit ist dann auch die Gültigkeit und Lebensdauer festgelegt)

Angabe des Ergebniswertes

- Eine Methode muss einen Ergebniswert von dem Typ liefern, der im Methodenkopf angegeben wurde

- Mit einer return-Anweisung gibt man Berechungsformel für Ergebniswert an
- Die Ausführung einer return-Anweisung beendet **sofort** die Ausführung der Methode und die Programmausführung wird an der Stelle fortgesetzt, an der die Methode aufgerufen wurde (gleich mehr)
- Es kann auch mehrere return-Anweisungen in einer Methode geben

- Zur Erinnerung:



- Spezialfall:
  - Ergebnistyp `void` bedeutet, dass diese Methode keinen Ergebniswert liefert
  - Dies ist sinnvoll, wenn in der Methode nur Seiteneffekte beabsichtigt sind und ein Ergebniswert im eigentlichen Sinne nicht anfällt
  - Beispiel: in Methode ist nur der Aufruf `System.out.println(...);`

Zwischenstand

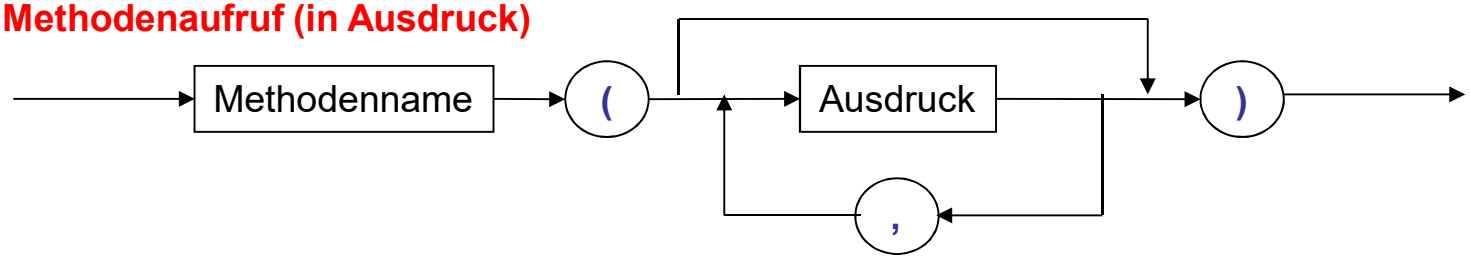
- Methoden stellen ein Strukturierungsmittel für Programme bereit
- Die nach außen / für einen Nutzer sichtbare Schnittstelle einer Methode wird im Methodenkopf angegeben
- Die meist nur für den Implementierer relevante Realisierung der Methode / Berechnung wird im Methodenrumpf angegeben

Reflektion

- Geben Sie einen analogen Java-Methodenkopf zur mathematischen Funktion f : N x B $\rightarrow$ R (es sei: N = natürliche Zahlen, B = Wahrheitswerte, R = reelle Zahlen).

2) Aufruf einer Methode

Methodenaufruf (in Ausdruck)



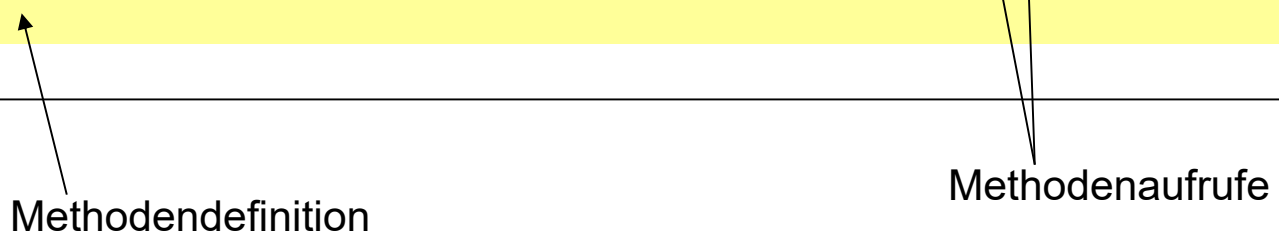
- Eine Methode lässt sich nutzen in einem Methodenaufruf, der formal ein Ausdruck ist und damit bei Auswertung einen Wert liefert
- Die Anzahl der aktuellen Parameter muss mit der Anzahl der formalen Parameter der Methodendefinition übereinstimmen
- Bei den Parametertypen ist möglich
  - Der Typ des aktuellen Arguments ist vom geforderten Typ
  - Der Typ des aktuellen Arguments kann durch eine (automatisch durchgeführte) erweiternde Typumwandlung in den geforderten Typ der Methodendefinition umgewandelt werden
- Ein Methodenaufruf wird (in Java) ausgewertet, indem
  - zuerst alle aktuellen Parameter von links nach rechts ausgewertet werden
  - die Methode angewandt wird
  - die Methode einen Wert als Resultat liefert

Beispiel

```
/**
 * Flaechenberechnung von Rechtecken
 */
public class Flaechenberechnung {

    public static void main(String[] args) {
        // Hier erfolgen zwei Aufrufe der Methode zur Flaechenberechnung
        System.out.println("Flaeche von Rechteck a ist " + flaeche(10.0, 10.0));
        System.out.println("Flaeche von Rechteck b ist " + flaeche(20.0, 5.0));
    }

    // Hier steht der Algorithmus zur Berechnung der Flaeche
    public static double flaeche(double laenge, double breite) {
        return laenge * breite;
    }
}
```



Zwischenstand

- In einem Methodenrumpf kann an jeder Stelle durch eine return-Anweisung die Methodenberechnung beendet werden und ein Ergebnis dieses Methodenaufrufs angegeben werden
- Bei einem Methodenaufruf werden die Argumentausdrücke von links nach rechts ausgewertet und dann erst die Methode mit diesen Werten aktiviert
- Nach Beenden einer Methodenberechnung wird die Ausführung an der Stelle fortgesetzt, von der der Aufruf erfolgte

Reflektion

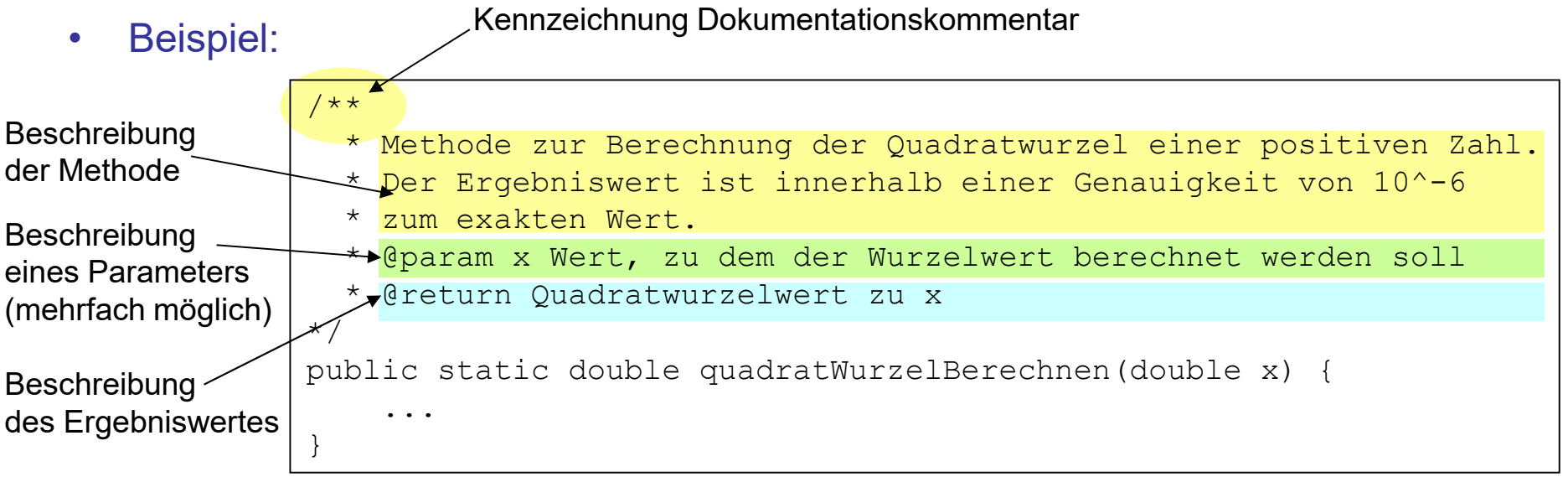
- Geben Sie eine Java-Methode zur mathematischen Funktion f : N x B $\rightarrow$ R an mit:

$$f(x,b) = \begin{cases} x & \text{falls b} \\ \text{x} + 0.5 & \text{sonst} \end{cases}$$

Dokumentation einer Methode

- Für Java-Methoden gibt es spezielle Dokumentationskommentare, die zur (informellen) Dokumentation der Schnittstelle dienen
- Mit Hilfe des javadoc Programms (Teil des Java SDK) lässt sich daraus ein HTML-basiertes Dokument generieren

- Beispiel:




```
/**
  * Methode zur Berechnung der Quadratwurzel einer positiven Zahl.
  * Der Ergebniswert ist innerhalb einer Genauigkeit von 10^-6
  * zum exakten Wert.
  * @param x Wert, zu dem der Wurzelwert berechnet werden soll
  * @return Quadratwurzelwert zu x
  */
public static double quadratWurzelBerechnen(double x) {
    ...
}
```

Überladen von Methoden

- Ebenso wie bei Operatoren gibt es auch bei Methoden die Möglichkeit des Überladens eines Methodenamens
- Zur Unterscheidung mehrerer Methoden mit gleichem Namen dient die **Signatur** einer Methode, wozu gehört:
  - Name der Methode
  - Anzahl der Parameter
  - Typ der Parameter inklusive Reihenfolge
- Die Signaturen aller Methoden einer Klasse müssen verschieden sein

- Das Überladen eines Methodennamens kann dann sinnvoll eingesetzt werden, wenn die gleiche/ähnliche Funktionalität für mehrere Typen / Typkombinationen oder für eine unterschiedliche Anzahl an Argumenten angeboten werden soll
- Beispiel: Substring erzeugen
  - `"abc".substring(von, bis)`
  - `"abc".substring(von)`

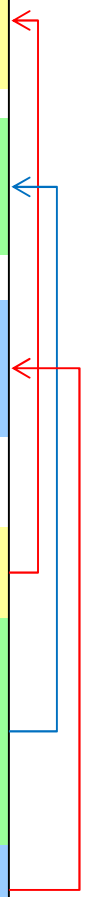
Beispiel 1

```
public class Ueberladen {
    public static void ausgeben(int x) {
        System.out.println("int-Wert ist " + x);
    }

    public static void ausgeben(double x) {
        System.out.println("double-Wert ist " + x);
    }

    public static void ausgeben(int x, double y) {
        System.out.println("int-Wert ist " + x + ", double-Wert ist " + y);
    }

    public static void main(String[] args) {
        // Aufruf mit einem int-Argument
        ausgeben(5);
        // Aufruf mit einem double-Argument
        ausgeben(5.0);
        // Aufruf mit einem float-Argument.
        // durch erweiternde Typanpassung wird double-Variante aufgerufen
        ausgeben(5.0f);
        // Aufruf mit einem int- und double-Argument
        ausgeben(3, 5.0);
    }
}
```



Beispiel 2 (Wiederholung)

- Wir kennen bereits das Verfahren von Heron zur Berechnung der Quadratwurzel eines Wertes
- Wir wenden wiederholt die Formel an: $y_{n+1} = \frac{y_n + \frac{x}{y_n}}{2}$

  und brechen die Berechnung ab, wenn wir "nahe genug" an den korrekten Wert gelangt sind
- Das ist der Fall, wenn $|y_n^2 - x| < \varepsilon$ für ein ε unserer Wahl
- Je kleiner ε gewählt ist, umso genauer werden wir mit dem Ergebniswert, aber umso länger muss auch iteriert werden

- Es gibt erfahrene Nutzer dieses Verfahrens, die ein geeignetes ε selbst bestimmen können und wollen
- Es gibt weniger erfahrene Nutzer dieses Verfahrens, die ε selbst nicht abschätzen können und deshalb lieber eine Vorgabe haben
- Lösung: überladene Methode mit zwei Versionen

Beispiel 2 (Teil 1)

```
public class QuadratWurzelBerechnung2 {
    public static void main(String[] args) {
        // berechne Quadratwurzel von 8157 mit einer Genauigkeit von 10^-6
        double wurzel1 = quadratWurzelBerechnen(8157.0, 1e-6);
        // berechne Quadratwurzel von 8157 mit "eingebauter" Genauigkeit
        double wurzel2 = quadratWurzelBerechnen(8157.0);
    }

    /**
     * Methode zur Berechnung der Quadratwurzel einer positiven Zahl.
     * Der Ergebniswert ist innerhalb einer Genauigkeit von 10^-6
     * zum exakten Wert.
     * @param x Wert, zu dem der Wurzelwert berechnet werden soll
     * @return Quadratwurzelwert zu x
     */
     public static double quadratWurzelBerechnen(double x) {
        // gewuenschte Genauigkeit
        double epsilon = 0.000001;

        // wir nutzen die andere Methode mit unserer Genauigkeitsvorgabe
        return quadratWurzelBerechnen(x, epsilon);
     }

    // naechste Folie geht es weiter
}
```

Beispiel 2 (Teil 2)

```
public class QuadratWurzelBerechnung2 {
   // siehe letzte Folie

   /**
     * Methode zur Berechnung der Quadratwurzel einer positiven Zahl.
     * @param x Wert, zu dem der Wurzelwert berechnet werden soll
     * @param epsilon Vorgabe der Berechnungsgenauigkeit
     * @return Quadratwurzelwert zu x
     */
    public static double quadratWurzelBerechnen(double x, double epsilon) {

        double fehler;      // Fehlerabschaetzung zum korrekten Wert
        double y = x;       // derzeitiger Annaeherungswert y_n

        do {
            // Berechnungsvorschrift fuer bessere Annaeherung
            y = 0.5 * (y + x/y);
            // Absolutbetrag fuer Fehlerabschaetzung
            fehler = Math.abs(x - y*y);
        } while(fehler > epsilon);

        // Ergebnis liefern
        return y;
     }
}
```

Zusammenfassung

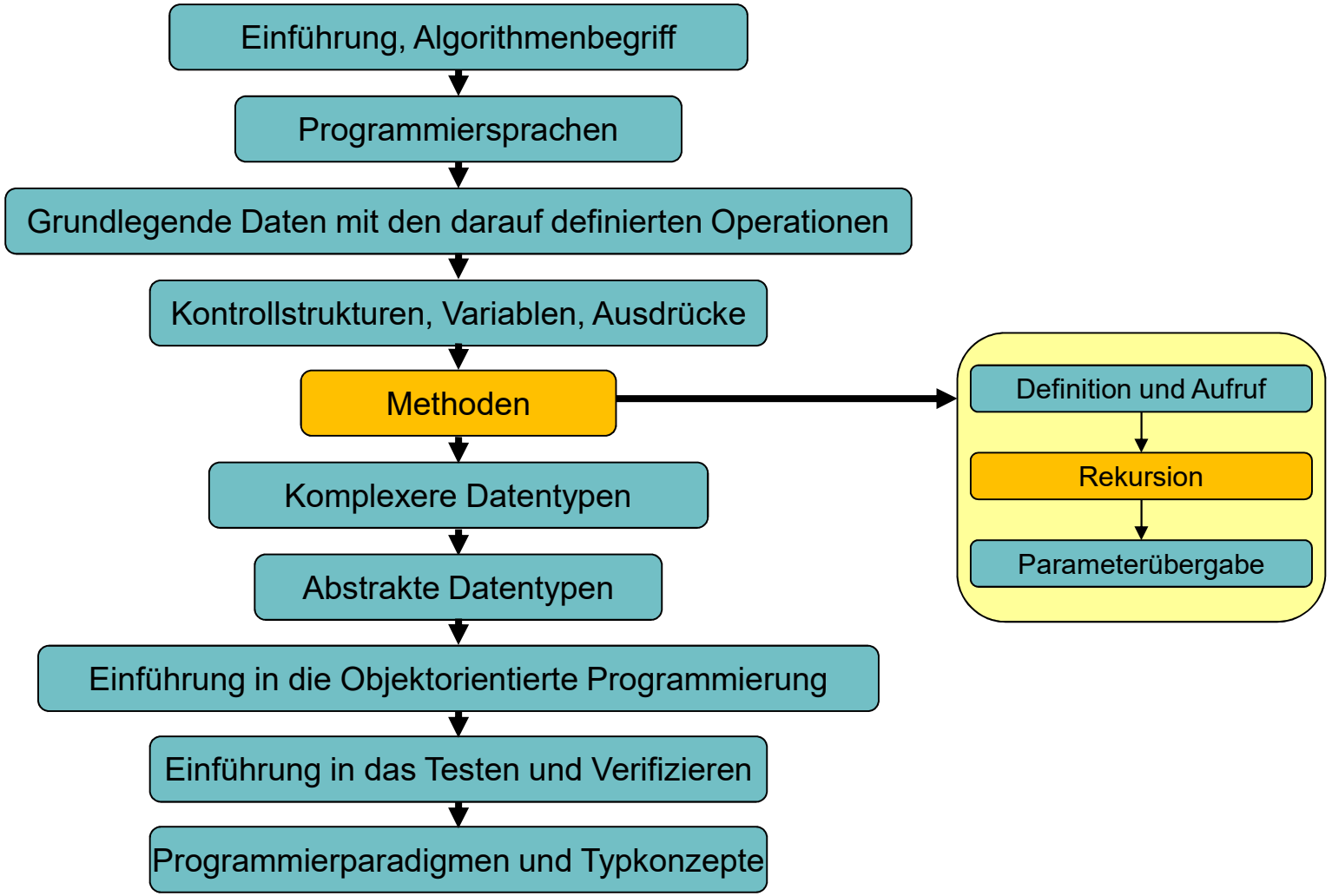
- Java bietet mehrere Möglichkeiten der Strukturierung von (großen) Programmen (Blöcke, Methoden, Klassen, Pakete)

- Methoden entsprechen den Funktionen in der Mathematik und werden ähnlich definiert (Name, Definitions- und Wertebereich, formale Parameter) und genutzt (Angabe aktueller Parameter im Aufruf)

- Es gibt genau eine Methodendefinition und beliebig viele Methodenaufrufe
- Eine Methodendefinition hat einen Methodenkopf (definiert Schnittstelle) und einen Methodenrumpf (Implementierung der Methode)

- In einem Methodenaufruf muss die Argumentanzahl und die geforderten Typen beachtet werden
- Methoden können überladen werden

Inhalt dieser Veranstaltung



Rekursive Funktionsdefinition

- Bekannt:

  flaeche : $R \times R \rightarrow R$

  flaeche(Länge,Breite) := Länge $\cdot$ Breite

  flaeche(10,20) = $10 \cdot 20 = 200$

- Was ist mit (auch als n! geschrieben):

  fakultaet : $N \rightarrow N$

  fakultaet(n) := $\begin{cases} 1 & \text{falls } n=0 \\ n \cdot \text{fakultaet}(n-1) & \text{sonst} \end{cases}$

  fakultaet(2) = $2 \cdot \text{fakultaet}(1) = 2 \cdot 1 \cdot \text{fakultaet}(0) = 2 \cdot 1 \cdot 1 = 2$

- Dies ist eine rekursiv definierte Funktion
- "rekursiv" laut Duden: "zurückgehend bis zu bekannten Werten"
- In der Mathematik und in Java kein Problem (wenn die Definition Sinn macht)!

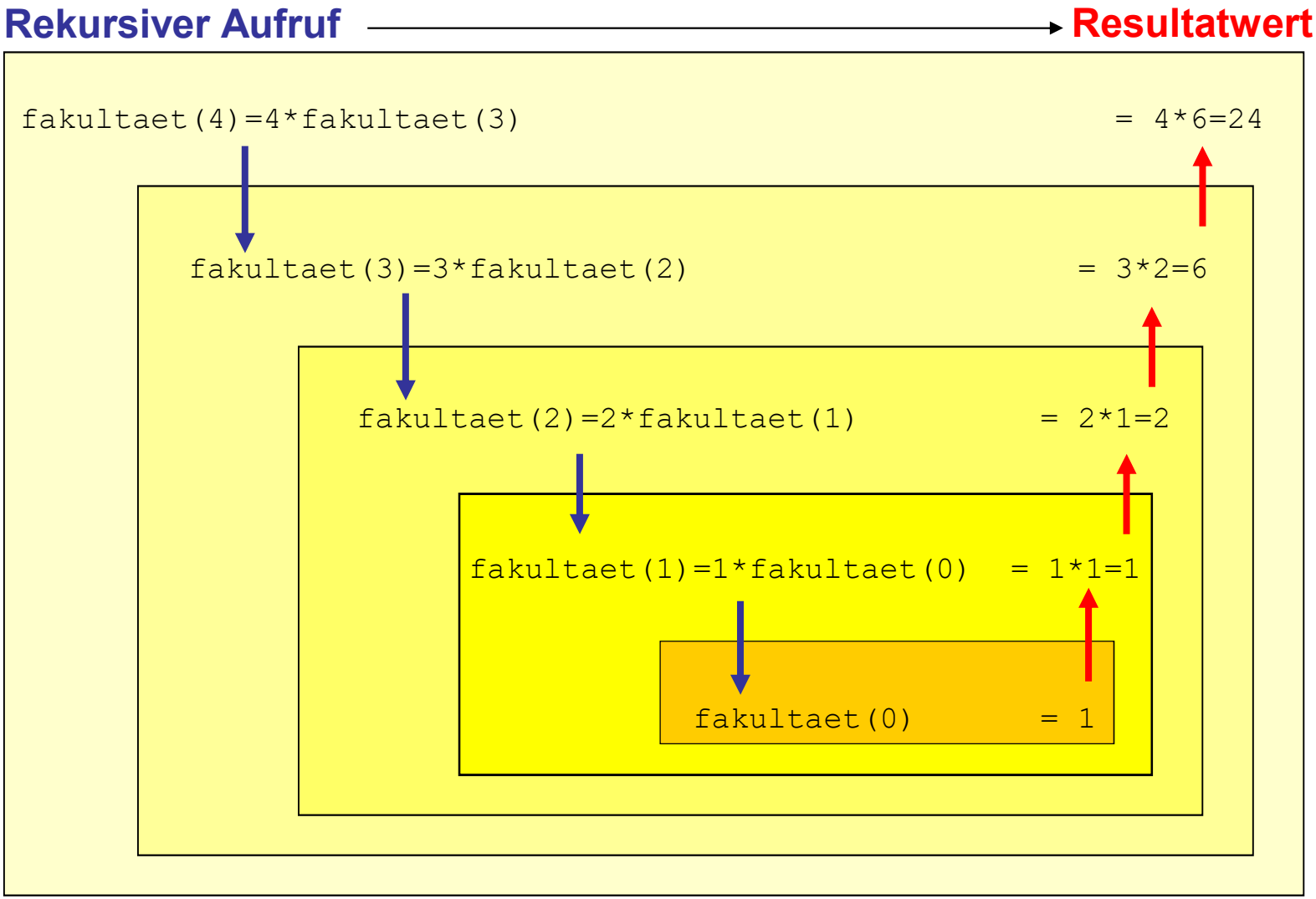
In Java

```
public class RekursiveMethoden {

    public static void main(String[] args) {
        // in einer Schleife Methodenwerte berechnen
        for(int i=0; i<10; i++) {
            // Aufruf Fakultaetsfunktion mit verschiedenen Argumenten
            System.out.println("fakultaet("+i+")=" + fakultaet(i));
        }
    }

    /**
     * Fakultaetsfunktion
     * @param n Argument
     * @return n!
     */
    public static int fakultaet(int n) {
        if (n == 0) {
            return 1;
        } else {
            return n * fakultaet(n-1);
        }
    }
}
```

Auswertung `fakultaet(4)`



Rekursive Definition

- Wie sind rekursive Funktionen aufgebaut?
- Ein oder mehrere Basisfälle (Beispiel: n=0)
- Ein oder mehrere allgemeine Fälle, die auf "kleinere" gleichartige Fälle zurückführen (Beispiel: n $\rightarrow$ n-1)

- Ein Problem existiert, wenn für mindestens ein Argument die Berechnung des Resultatwertes nicht zu einem Abbruch der Auswertung führt
- Beispiele:
  - `f : N` $\rightarrow$ `N` mit `f(n)=f(n+1)`
  - `g : N` $\rightarrow$ `N` mit

$$g(n) := \begin{cases} 0 & \text{falls } n=0 \\ g(n-2) & \text{falls } n \text{ gerade} \\ g(n+2) & \text{falls } n \text{ ungerade} \end{cases}$$

- Später: Rekursion als allgemeines Problemlösungskonzept anwenden

Weitere Beispiele für rekursive Definitionen

- Fibonacci-Funktion $\texttt{fib} : \texttt{N} \rightarrow \texttt{N}$ mit

$$\texttt{fib(n)} := \begin{cases} \texttt{0} & \texttt{falls n=0} \\ \texttt{1} & \texttt{falls n=1} \\ \texttt{fib(n-1)+fib(n-2)} & \texttt{sonst} \end{cases}$$

- Binomialkoeffizient $\binom{n}{k} : \texttt{N} \times \texttt{N} \rightarrow \texttt{N}$ mit

$$\binom{n}{k} := \begin{cases} 0 & \text{falls } n < k \\ 1 & \text{falls } k=0 \text{ oder } n=k \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{sonst} \end{cases}$$

In Java

```
public static int fibonacci(int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fibonacci(n-2) + fibonacci(n-1);
    }
}

public static int binomial(int n, int k) {
    if (n < k) {
        return 0;
    } else if ((k == 0) || (n == k)) {
        return 1;
    } else {
        return binomial(n-1, k-1) + binomial(n-1, k);
    }
}
```

Indirekte Rekursion und Mehrfachrekursion

- Indirekte Rekursion

  `f : N → N` mit `f(x) = g(x+1)`

  `g : N → N` mit `g(x) = f(x-2)` falls `x > 1` und `0` sonst

- Mehrfachrekursion (Beispiel Ackermann-Funktion)

  `ack : N × N → N` mit

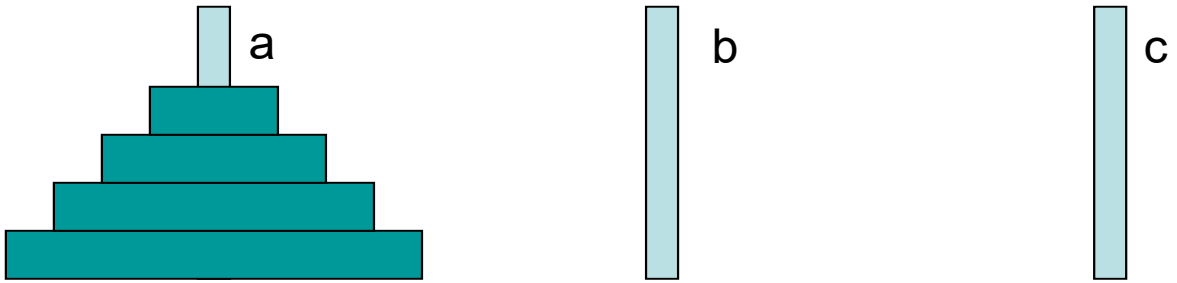
$$\texttt{ack(m,n)} := \begin{cases} \texttt{n+1} & \texttt{falls m=0} \\ \texttt{ack(m-1,1)} & \texttt{falls n=0} \\ \texttt{ack(m-1, ack(m,n-1))} & \texttt{sonst} \end{cases}$$

- Alles kein Formulierungsproblem in Java!

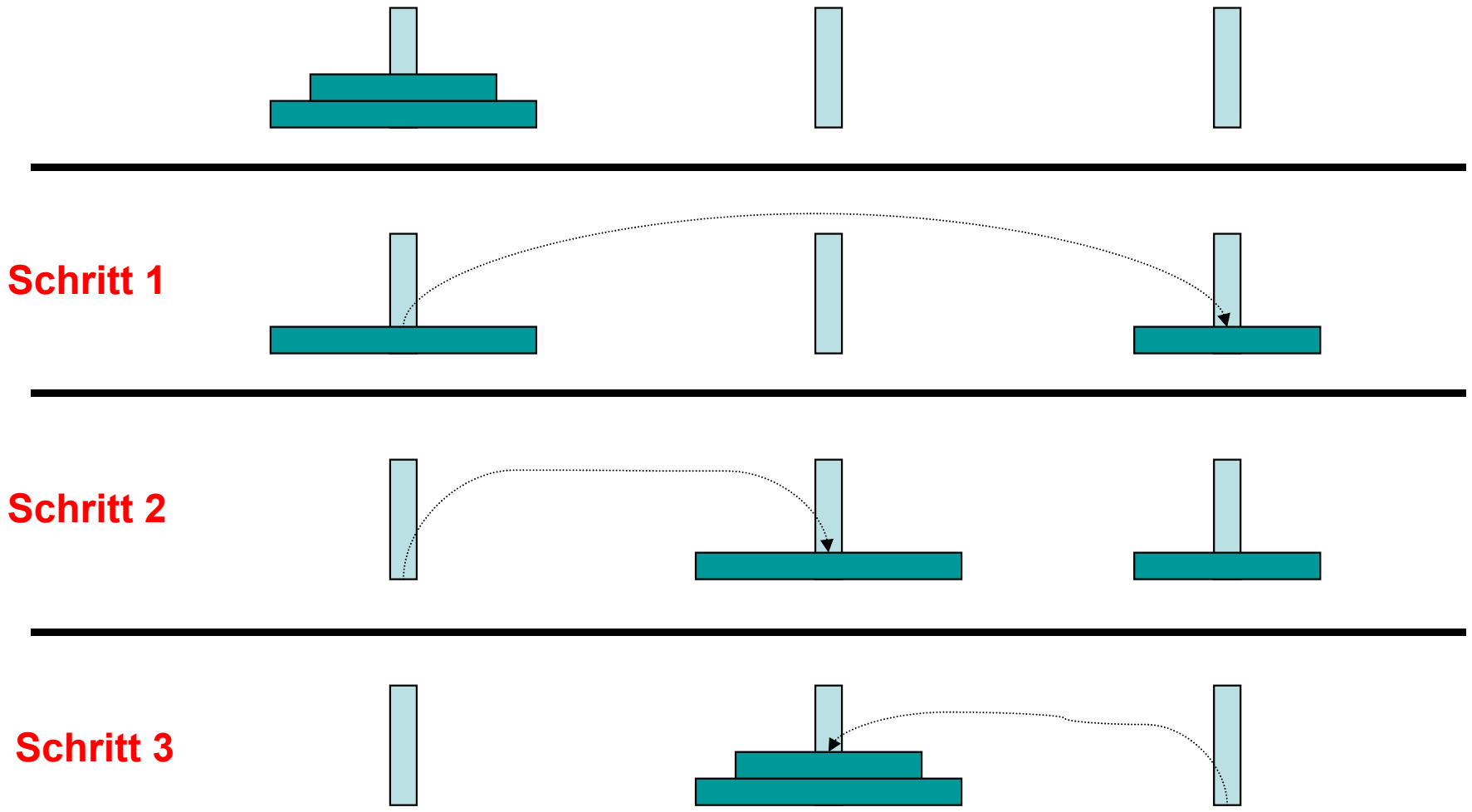
- Ggfs. Spezialfälle zu Beginn eines Verfahrens in einer nicht-rekursiven Einstiegsmethode behandeln und anschließend die eigentliche rekursive Behandlung in einer eigenen Methode anwenden

Türme von Hanoi

- Der Legende nach befasste sich ein buddhistischer Mönchsorden in Hanoi mit der folgenden Aufgabe.
  - Auf einem Brett sind 3 senkrechte Stäbe (mit a,b und c bezeichnet) und n runde Scheiben verschiedener Größe, die in der Mitte ein rundes Loch haben, so dass man die Stäbe dadurch stecken kann.
  - Zu Beginn des Spieles sind alle Scheiben auf Stab a aufgesteckt (größte Scheibe unten, kleinste Scheibe oben).
  - Die Aufgabe ist es, den Scheibenturm von a nach b zu bekommen.
  - Einschränkung: Man darf immer nur die oberste Scheibe eines Stapels bewegen und nie darf eine größere Scheibe auf einer kleineren Scheibe liegen.



Lösungsbeispiel für n=2



Erster Ansatz einer Lösungsstrategie

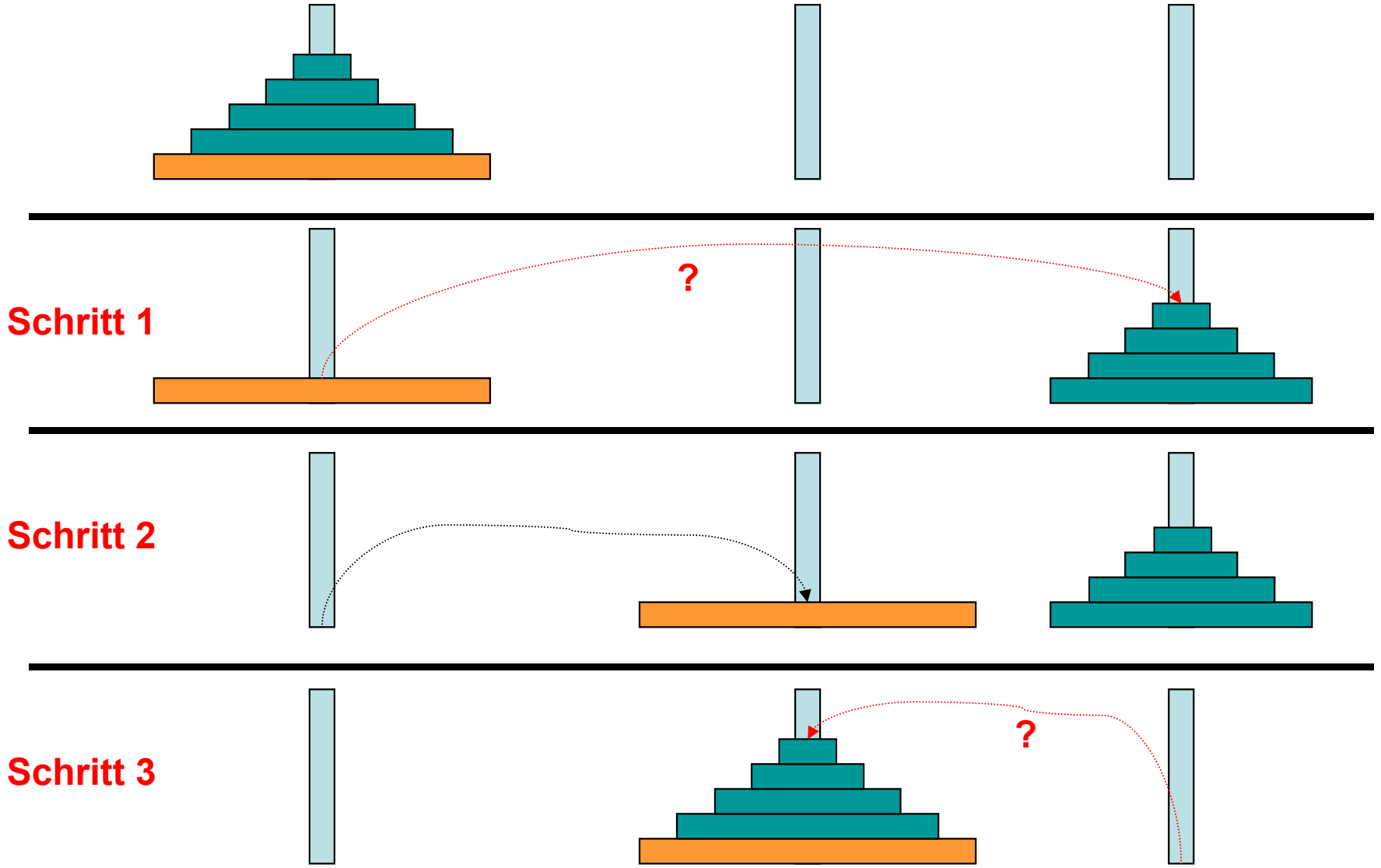
- Aufgrund der beiden Einschränkungen ist klar, dass für beliebige Stapel die einzige Möglichkeit, die größte zuunterst liegende Scheibe von a nach b zu bekommen, darin liegt, alle restlichen Scheiben von a wegzuschaffen. Dies kann nur dadurch geschehen, dass man die restlichen Scheiben auf c bringt.

- Ein erster Lösungsansatz für einen Stapel mit n Scheiben ist also:
  - Übertrage die obersten n-1 Scheiben von a nach c
  - Bewege unterste Scheibe von a nach b
  - Übertrage die obersten n-1 Scheiben von c nach b

- Wir lassen zuerst offen, wie das Übertragen eines Teilstapels aussieht.

- Wir unterscheiden:
  - Übertragen: Übertragen eines (Teil-) Stapels
  - Bewegen: Elementarschritt mit genau einer Scheibe

Beispiel



Allgemeiner Lösungsansatz

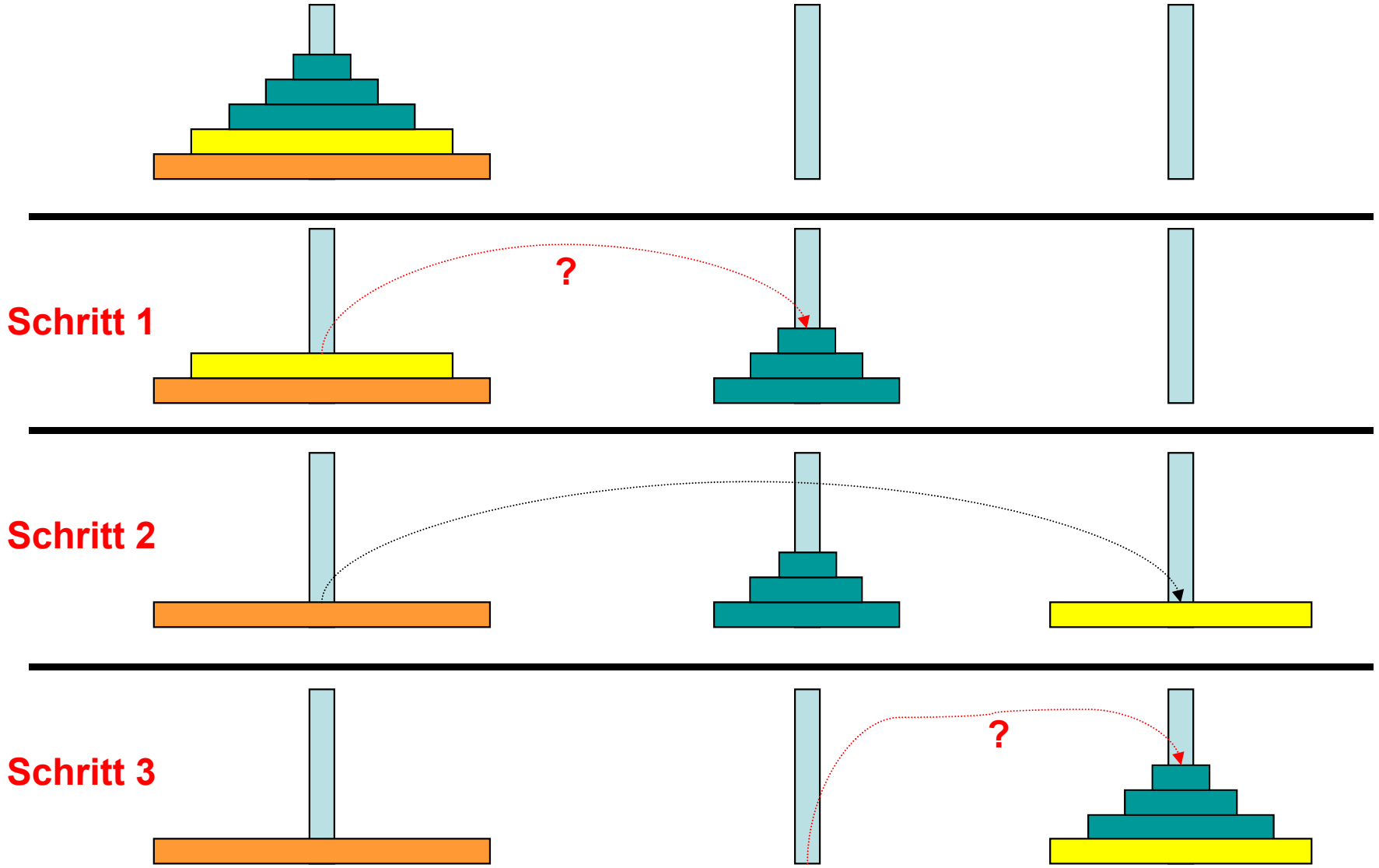
- Ein erster Lösungsansatz für einen Stapel mit n Scheiben war:
  - Übertrage die obersten n-1 Scheiben von a nach c
  - Bewege unterste Scheibe von a nach b
  - Übertrage die obersten n-1 Scheiben von c nach a

- Bezeichnet man mit Quelle den Stab, wo ursprünglich der Stapel liegt, mit Senke den Stab, wo der Stapel hin soll und mit Arbeitsbereich den Stab, auf den der Teilstapel temporär abgelegt werden soll, kann man den ersten Lösungsansatz formulieren:

```
Einzelschritt (Quelle, Senke, Arbeitsbereich, n)
    Übertragen(Quelle, Arbeitsbereich, n-1)      // ? noch offen
    Bewege(Quelle, Senke)                        // Elementarschritt
    Übertragen(Arbeitsbereich, Senke, n-1)       // ? noch offen
```

- Unser konkretes Problem, das Übertragen des Stapels von a nach b, würde also heißen: Einzelschritt(a,b,c,n)

Beispiel



Allgemeine Lösung

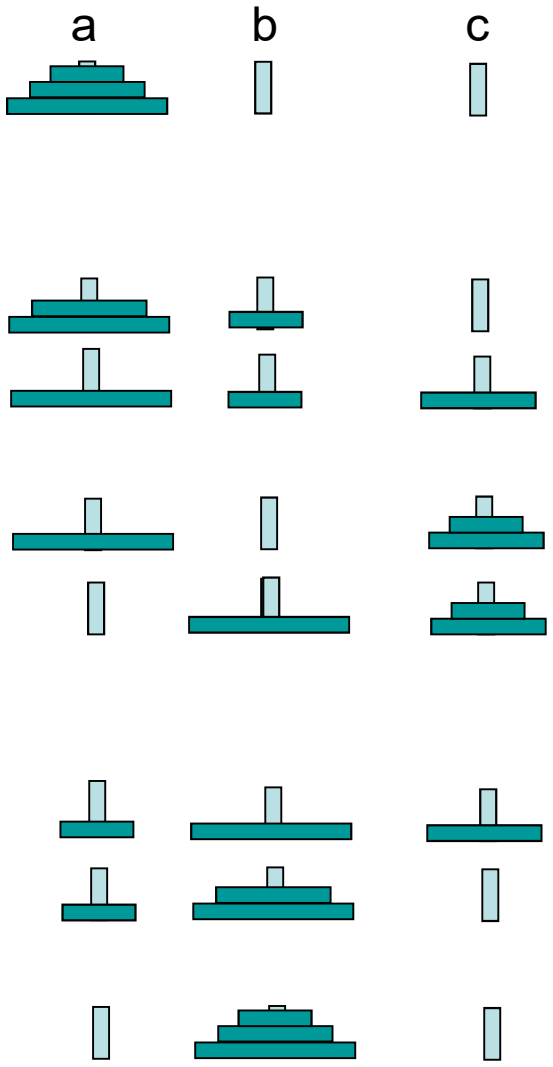
- Wir können also eine Gesamtlösung bekommen, in dem wir den eben angegebenen Algorithmus rekursiv auf Teilprobleme anwenden.
- Die Rekursion bricht ab, wenn ein Stapel mit genau einer Scheibe vorliegt.

- Gesamtalgorithmus:

```
Hanoi(Quelle, Senke, Arbeitsbereich, n)
    Falls n=1
            Bewege(Quelle, Senke)                          // Elementarschritt
    Sonst
            Hanoi(Quelle, Arbeitsbereich, Senke, n-1)      // rekursiv
            Bewege(Quelle, Senke)                          // Elementarschritt
            Hanoi(Arbeitsbereich, Senke, Quelle, n-1)      // rekursiv
```

Ausführung für n=3

```
Hanoi(a,b,c,3)
    Quelle=a, Senke=b, Arbeit=c, n=3
    Falls 3==1 dann... Sonst Hanoi(a,c,b,2)
        Quelle=a, Senke=c, Arbeit=b,n=2
        Falls 2==1 dann..Sonst Hanoi(a,b,c,1)
            Quelle=a, Senke=b, Arbeit=c, n=1
            Falls 1==1 dann bewege(a,b)
        bewege(a,c)
        Hanoi(b,c,a,1)
            Quelle=b, Senke=c, Arbeit=a, n=1
            Falls 1==1 dann bewege(b,c)
    bewege(a,b)
    Hanoi(c,b,a,2)
        Quelle=c, Senke=b, Arbeit=a, n=2
        Falls 2==1 dann..Sonst Hanoi(c,a,b,1)
            Quelle=c, Senke=a, Arbeit=b, n=1
            Falls 1==1 dann bewege(c,a)
        bewege(c,b)
        Hanoi(a,b,c,1)
            Quelle=a, Senke=b, Arbeit=c, n=1
            Falls 1==1 dann bewege(a,b)
```



Hanoi als Programm

```
public class Hanoi {

    public static void main(String[] args) {
        int n=3;

        // bewege Stapel mit n=3 Scheiben von Stab 0 nach Stab 1 über Stab 2
        Hanoi(0,1,2,n);
    }

    public static void Hanoi(int Quelle, int Senke, int Arbeitsbereich, int n) {
        if(n == 1)
            bewege(Quelle, Senke);
        else {
            Hanoi(Quelle, Arbeitsbereich, Senke, n-1);
            bewege(Quelle, Senke);
            Hanoi(Arbeitsbereich, Senke, Quelle, n-1);
        }
    }

    public static void bewege(int Quelle, int Senke) {
        System.out.println("Bewege Scheibe von " + Quelle + " nach " + Senke);
    }
}
```

Berechnungsaufwand

- Jeder Aufruf von Hanoi, der in den else-Fall läuft, bewirkt zwei weitere Aufrufe von Hanoi, d.h. zwei Berechnungen für n, jeder dieser Aufrufe wiederum 2 usw. Insgesamt ergeben sich damit 2^n-1 Aufrufe von Hanoi.
- In jedem Aufruf wird genau einmal (direkt) eine Scheibe bewegt (im then-Fall oder im else-Fall). Insgesamt ergeben sich damit 2^n-1 Scheibenbewegungen.

- Das Ursprungsproblem soll für n=64 formuliert gewesen sein. Angenommen, die Mönche hätten in jeder Sekunde eine Scheibe bewegt (ohne Fehler), so wären die Mönche 2^{64} - 1 s $\approx$ 600.000.000.000 Jahre beschäftigt gewesen.

Zwischenstand

- Rekursive Funktionsdefinitionen sind in Java möglich, ohne dass die Rekursion explizit behandelt werden müsste
- Die Auswertung geschieht analog einer „normalen“ Methodenauswertung
- Das Problem der Türme von Hanoi lässt sich mit einem rekursiven Lösungsansatz lösen
- Die Rechenaufwand / die Komplexität dieses Verfahrens ist sehr hoch

Reflektion

- Werten Sie auf dem Papier aus: `binomial(2,1)`
- Könnte man das Problem der Türme von Hanoi auch iterativ lösen? Wenn nein, wieso nicht? Wenn ja, was wäre der prinzipielle Ansatz dazu?

Fakultät in zwei Versionen

rekursive Formulierung

```
public static int fakultaet(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fakultaet(n-1);
    }
}
```

iterative Formulierung

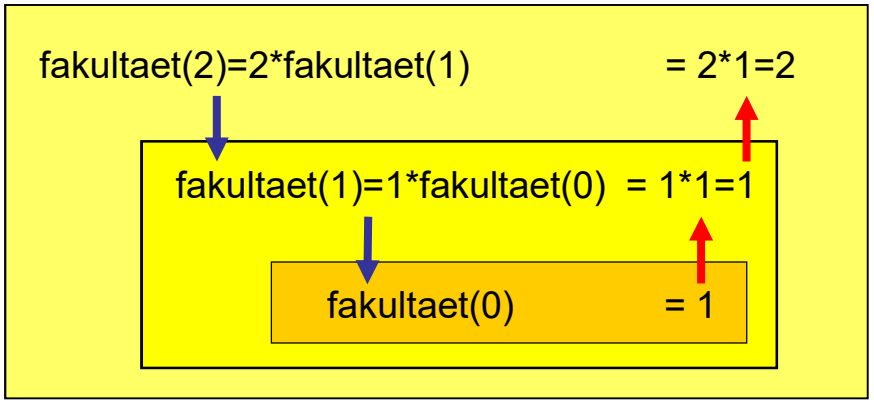
```
public static int fakultaet(int n) {
    int ergebnis = 1;
    for(int i=1; i<=n; i++) {
        ergebnis *= i;
    }
    return ergebnis;
}
```

- Fragen:
  - Ist eine Version besser als die andere (und nach welchen Kriterien)?
  - Kann ich immer und nach einem festen Algorithmus eine rekursive Version in eine iterative Version umwandeln und umgekehrt?

Vor- und Nachteile rekursiver Lösungen

- Vorteil: oft ist eine kompakte und natürliche Lösungsformulierung möglich (führe allgemein komplexe Fälle auf kleinere gleichartige Fälle zurück)

- Nachteil:
  - Wie wir später im Detail sehen werden, müssen noch offene Teilberechnungen verwaltet werden
  - Diese Verwaltung kann Zeit und (eventuell erheblich) Speicherplatz benötigen
  - Iterative Lösungen haben oft einen geringeren Verwaltungsaufwand



Umwandlung iterativ nach rekursiv

- Wir beschränken uns hier auf eine while-Schleife. Die Umwandlung der anderen Schleifentypen in eine while-Schleife wurde bereits gezeigt.
- Jede while-Schleife lässt sich nach folgendem Schema in eine rekursive Methode umwandeln:

```
while (Bedingung)
    Anweisung
```

wird umgewandelt zu:

```
SchleifeRekursiv() {
    if(Bedingung) {
        Anweisung;
        SchleifeRekursiv();
    }
}
```

- Wir lassen den Aspekt des Gültigkeitsbereichs und der Lebensdauer von Variablen hier außen vor (dies ist aber auch lösbar)

Beispiel

```
public class SchleifeRekursiv {
    static int i, summe;
    public static void main(String[] args) {
        i = 0; summe = 0;
        Schleife();

        i = 0;  summe = 0;
        SchleifeR();
    }

    public static void Schleife() {
        while(i < 10) {
            summe = summe + i;
            i = i + 1;
        }
    }

    public static void SchleifeR() {
        if(i < 10) {
            summe = summe + i;
            i = i + 1;
            SchleifeR();
        }
    }
}
```

iterative Version

rekursive Version

Umwandlung rekursiv nach iterativ

- Die Umwandlung einer beliebigen rekursiv definierten Methode in eine Iteration ist generell und nach einem festen Algorithmus nicht möglich

- Dies ist nur möglich, wenn eine endrekursive Methode vorliegt
- In einer endrekursiven Methode ist die letzte Aktion der Methodenauswertung der rekursive Aufruf

- Das Rekursionsschema (Muster) für eine einstellige endrekursive Funktion $f : N \rightarrow N$ lautet:

$$f(x) := \begin{cases} g(x) & \text{falls } P(x) \\ f(h(x)) & \text{sonst} \end{cases}$$

wobei $P : N \rightarrow B$ ein Prädikat ist (liefert einen Wahrheitswert) und $g, h : N \rightarrow N$ beliebige Funktionen

Umwandlung endrekursiv nach iterativ

- Zu dem Rekursionsschema $f : N \rightarrow N$ mit

$$f(x) := \begin{cases} g(x) & \text{falls } P(x) \\ f(h(x)) & \text{sonst} \end{cases}$$

- ist die äquivalente iterative Formulierung (in Java):

```
public static int f(int x) {
    while (!P(x)) {
        x = h(x);
    }
    return g(x);
}
```

Zwischenstand

- Rekursive Lösungen können einen (erheblichen) Verwaltungsaufwand bei der Auswertung eines Aufrufs nach sich ziehen
- Jede iterativ formulierte Lösung lässt sich nach einem festen Verfahren in eine äquivalente rekursive Lösung überführen
- Der umgekehrte Weg ist allgemein nicht möglich. Nur für endrekursive Formulierungen lässt sich ein festes Übersetzungsschema angeben.

Reflektion

- Rekursion ist also mächtiger als Iteration und jede iterative Formulierung lässt sich in eine rekursive Formulierung übersetzen, aber nicht umgekehrt. Wieso lässt man dann die Iteration als Programmierkonzept nicht komplett aus Programmiersprachen raus?

Zusammenfassung

- Rekursive Funktionsdefinitionen haben Basisfälle und allgemeine Fälle, die ein komplexes Problem auf ein weniger komplexes gleichartiges Problem zurückführen
- Solche rekursiven Definitionen sind auch in Java möglich
- Rekursive Lösungen in Programmen haben gegenüber iterativen Lösungen oft den Nachteil, dass die Verwaltung zusätzliche Ressourcen kostet (Zeit und Speicher)
- Iterative Lösungen lassen sich in rekursive umwandeln, umgekehrt ist dies nicht generell möglich