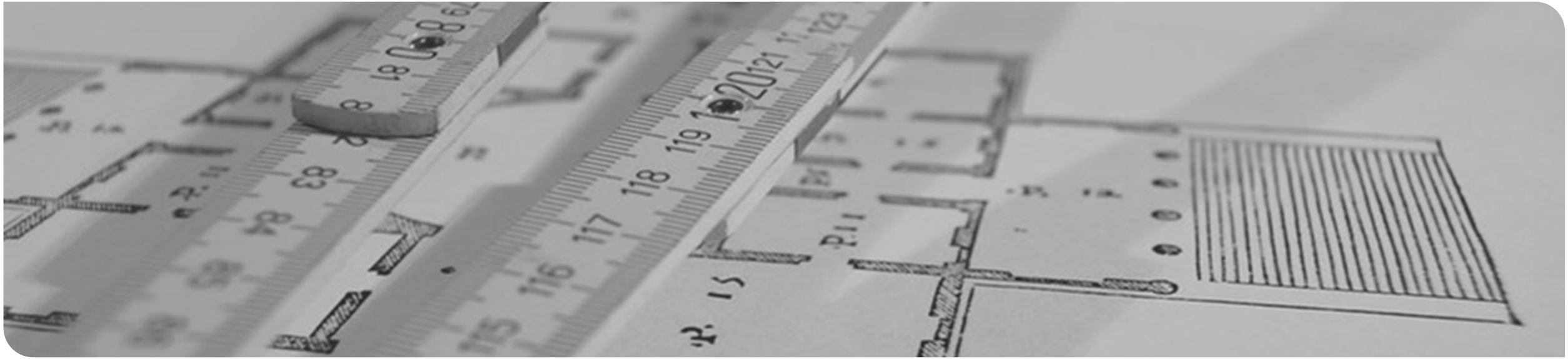
Karlsruher Institut für Technologie

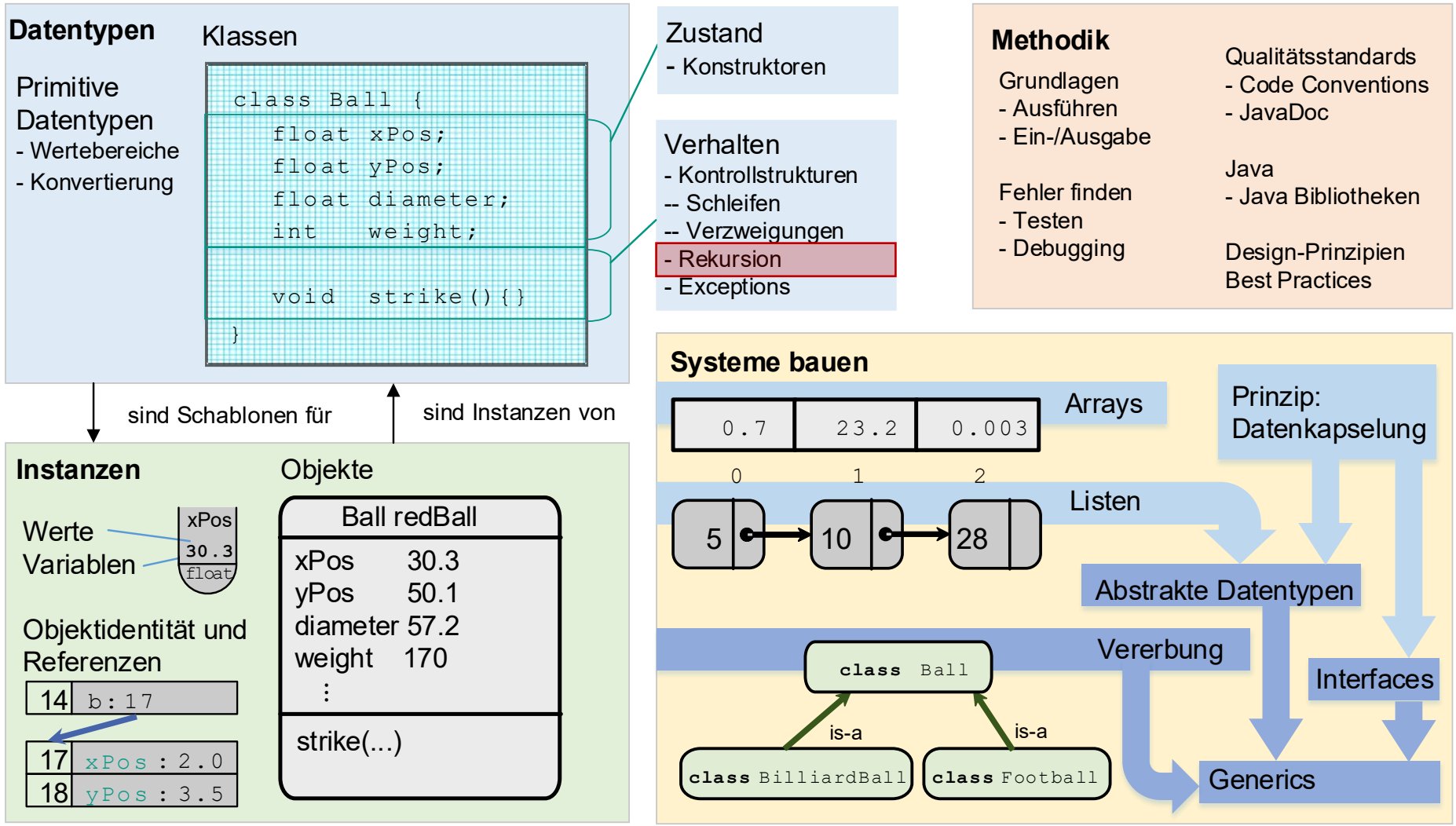
Vorlesung Programmieren

11. Rekursion

Prof. Dr.-Ing. Anne Koziolek



Vorlesungsüberblick: Objekt-orientiertes Programmieren in Java



Vorlesungsüberblick: Vorläufiger Semesterplan

21.10.2024	Erstsemesterbegrüßung: Einführung
23.10.2024	Organisatorisches; Ein Einfaches Programm, Objekte und Klassen
30.10.2024	Typen und Variablen
06.11.2024	Kontrollstrukturen (+Scanner)
13.11.2024	Konstruktoren und Methoden
20.11.2024	Arrays; Konvertierung, Datenkapselung, Sichtbarkeit
27.11.2024	Listen und Abstrakte Datentypen
04.12.2024	Vererbung
11.12.2024	Exceptions; Interfaces
18.12.2024	Generics; Rekursion
08.01.2025	Java-API; Objektorientierte Design-Prinzipien
15.01.2025	Best Practices; Finden und Beheben von Fehlern
22.01.2025	Testen und Assertions
29.01.2025	JUnit; Parsen, Suchen, Sortieren
05.02.2025	Vom Programm zur Maschine; Ausblick auf zukünftige Lehrveranstaltungen
12.02.2025	Wrap-Up

Lernziele

Rekursion

- Sie können rekursive Funktionen implementieren, mit denen Sie ein Problem in kleinere Instanzen zerteilen
- Sie können beschreiben, wie der Aufrufstapel (Call Stack) während der Abarbeitung von Methoden verwendet wird.

Quelle: http://phdcomics.com

Motivation: Divide and Conquer

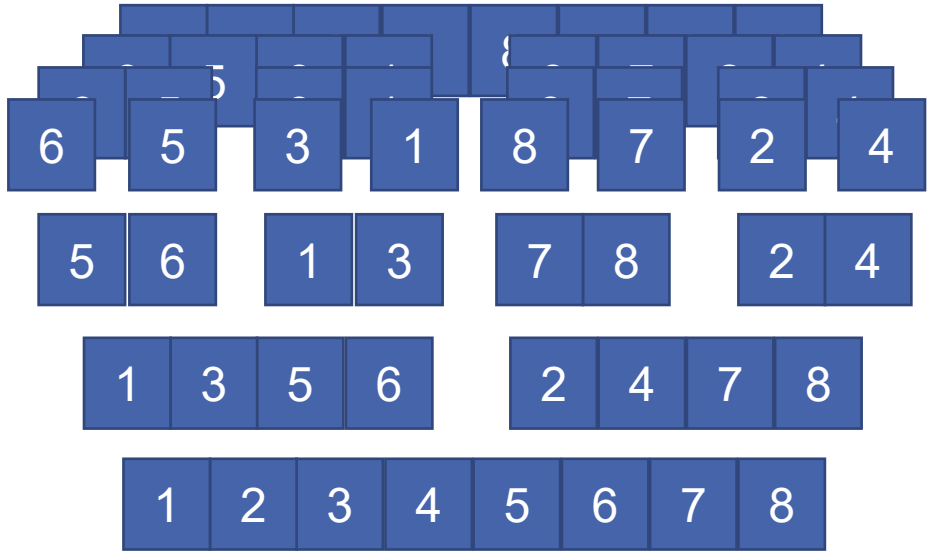
»divide et impera« / »teile und herrsche«

Wichtiges Grundprinzip der Algorithmik:

- Um ein Problem zu lösen, teile es in mehrere (einfachere) Teilprobleme auf
- Löse die einzelnen Teilprobleme
- Vereine die Teillösungen zu einer Gesamtlösung

Beispiel: Merge-Sort

1. Problem aufteilen durch Halbieren
2. Vergleiche Listenköpfe von jeweils zwei Listen
3. Kopiere kleineres Element in neue Liste (ans Ende)
4. Ergebnis: sortierte Listen
5. Wiederhole 2. – 4.



Rekursion

Prinzip der Rekursion

Man führe das gleiche Berechnungsmuster immer wieder mit einfacheren bzw. kleineren Eingabedaten aus, bis man zu einer trivialen Eingabe gelangt

Realisierung

Methoden, die sich direkt oder indirekt selbst aufrufen

Rekursion ist die Standard-Implementierung von Divide-and-Conquer

Rekursive Methoden

Definition

- Eine Methode `f` heißt (direkt) **rekursiv**, wenn im Rumpf von `f` Aufrufe von `f` vorkommen.
- Eine Methode `f` heißt **indirekt rekursiv**, wenn im Rumpf von `f` eine Methode `g` aufgerufen wird, die ihrerseits direkt oder indirekt auf Aufrufe von `f` führt.
- Eine Methode `f` heißt **endständig rekursiv**, wenn im Rumpf von `f` nach dem rekursiven Aufruf von `f` kein weiterer Code steht.

Bei jedem rekursiven Aufruf wird eine neue »Instanz« der jeweiligen Methode gestartet.

➜ Jede Instanz hat ihre eigenen lokalen Variablen und Parameter, welche »von außen« nicht sichtbar sind

Beispiel: Fakultätsfunktion

- Die Fakultätsfunktion n! berechnet das Produkt der Zahlen $1, 2, \ldots, n$
- Rekursiv lässt sich n! daher so berechnen:

$$0! = 1$$
$$n! = n \cdot (n-1)! \qquad \text{für } n \geq 1$$

Dies lässt sich einfach als Funktion in Java schreiben:

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

Beispiel: Fakultätsfunktion

Was passiert beim Aufruf von `fac(4)`?

```
fac(4)
= if (4 > 0) { 4 * fac(4-1) } else { 1 } )                          // Einsetzen
= 4 * fac(3)                                                        // Auswerten
= 4 * (if (3 > 0) { 3 * fac(3-1) } else { 1 } )                     // Einsetzen
= 4 * (3 * fac(2))                                                  // Auswerten
= 4 * (3 * (if (2 > 0) { 2 * fac(2-1) } else { 1 } ))               // Einsetzen
= 4 * (3 * (2 * fac(1)))                                            // Auswerten
= 4 * (3 * (2 * (if (1 > 0) { 1 * fac(1-1) } else { 1 } )))         // Einsetzen
= 4 * (3 * (2 * (1 * fac(0))))                                      // Auswerten
= 4 * (3 * (2 * (1 * ( if (0 > 0) { 1 * fac(0-1) } else { 1 } )))) // Einsetzen
= 4 * (3 * (2 * (1 * 1)))                                           // Auswerten
= 4 * (3 * (2 * 1))
= 4 * (3 * 2)
= 4 * 6
= 24
```

```java
public static int fac(int n) {
  if (n > 0) {
    return n * fac(n-1);
  } else {
    return 1;
  }
}
```

Binomialfunktion (I)

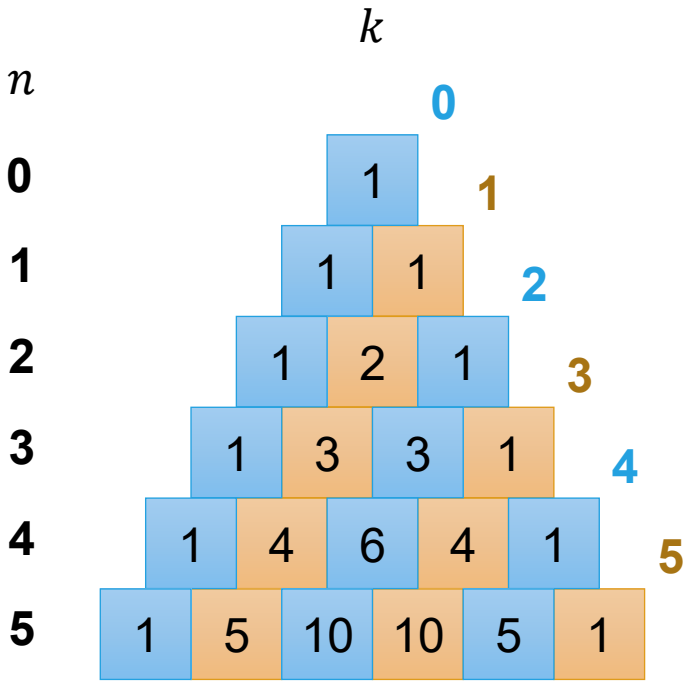
Die Binomialfunktion für $n \geq k \geq 0$ ist mit Hilfe der Fakultätsfunktion definiert als:

$$\binom{n}{k} := \frac{n!}{(n-k)! \cdot k!}$$

Eine direkte Implementierung ist **Overflow**-gefährdet, da die Fakultätsfunktion nur für kleine n im Rechner darstellbar ist.

Für die Binomialfunktion gilt aber:

$$\binom{n}{0} = 1, \qquad \binom{n}{n} = 1, \qquad \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$



Binomialfunktion (II)

Damit lässt sich die Binomialfunktion rekursiv berechnen:

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

```
public static int binom(int n, int k) throws IllegalArgumentException {
  if (k < 0 || k > n) {
    throw new IllegalArgumentException("binom(..) expects arguments n >= k >= 0");
  } else {
    if (k == 0 || k == n) {
      return 1;
    } else {
      return binom(n-1, k-1) + binom(n-1, k);
    }
  }
}
```

Diese nötige Fehlerbehandlung für Übersichtlichkeit auf den folgenden Folien weggelassen

Ausführung von binom(3,2)

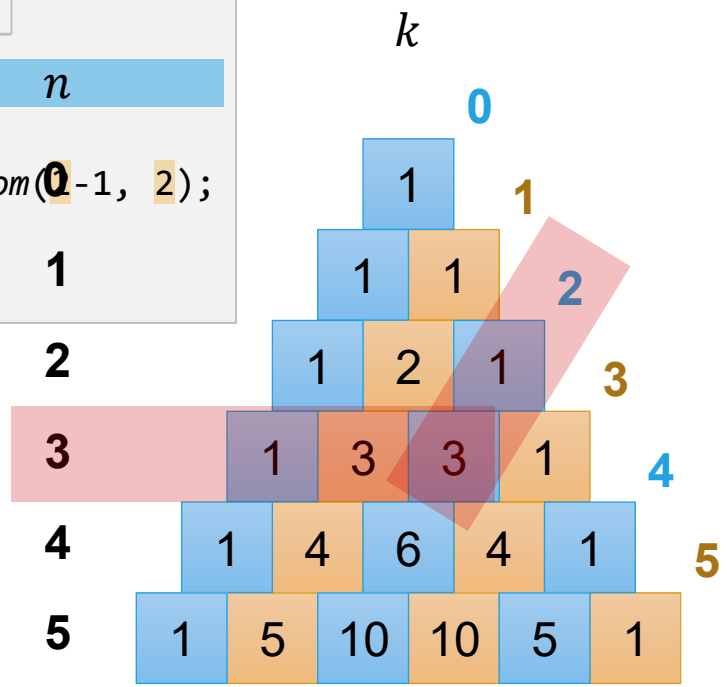
```
binom(int 3, int 2) {
  if (2 == 0 || 2 == 3) {
      return 1;
    } else {
      return 3;
  }
}
```

```
binom(int 2, int 1) {
  if (1 == 0 || 1 == 2) {
      return 1;
    } else {
      return 2;
  }
}
```

```
  if (1 == 0 || 1 == 1) {
      return 1;
    } else {
      return binom(1-1, 1-1) + binom(1-1, 1);
  }
}
```

```
      return 1;
    } else {
      return binom(2-1, 2-1) + binom(2-1, 2);
  }
}
```

binom(3,2) = 3



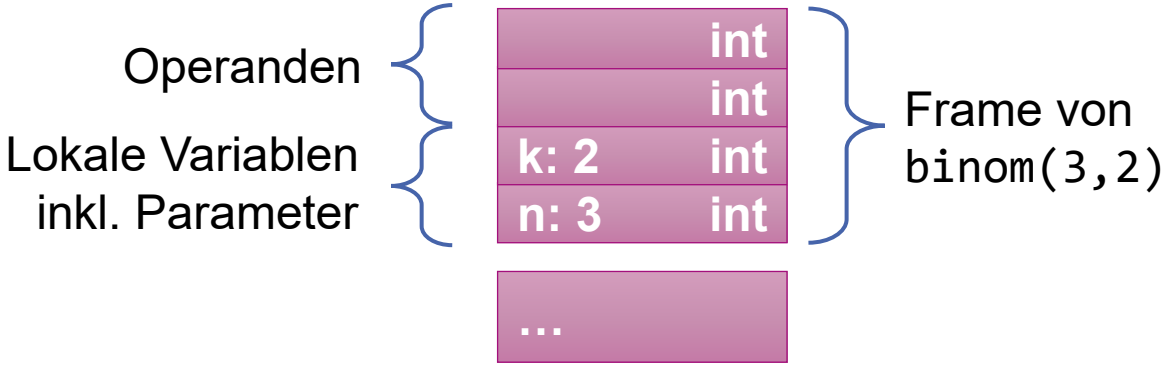
vgl. https://de.wikipedia.org/wiki/Binomialkoeffizient #Rekursive_Darstellung_und_Pascalsches_Dreieck

Methoden im Aufrufstapel

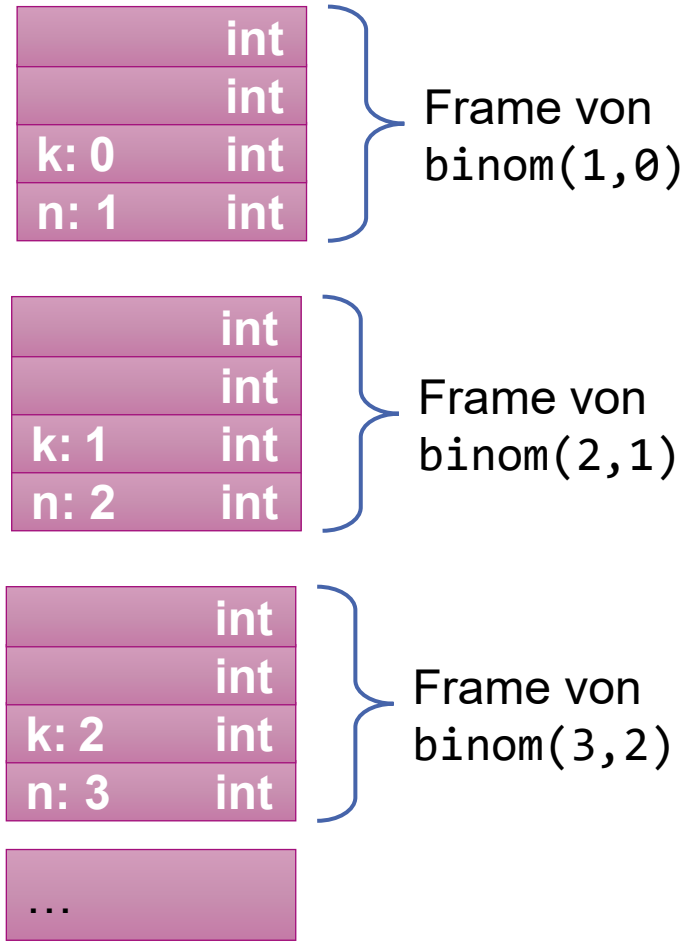
Lokale Variable (und Parameter) einer Methode werden im **Aufrufstapel** (*call stack* oder **Laufzeitkeller**) abgelegt

Für jede Methode wird ein neuer Speicherbereich (**Schachtel** bzw. *frame*) auf dem Aufrufstapel angelegt mit u.a.

- Lokalen Variablen (inkl. Parameter)
- Operanden



Methoden im Aufrufstapel

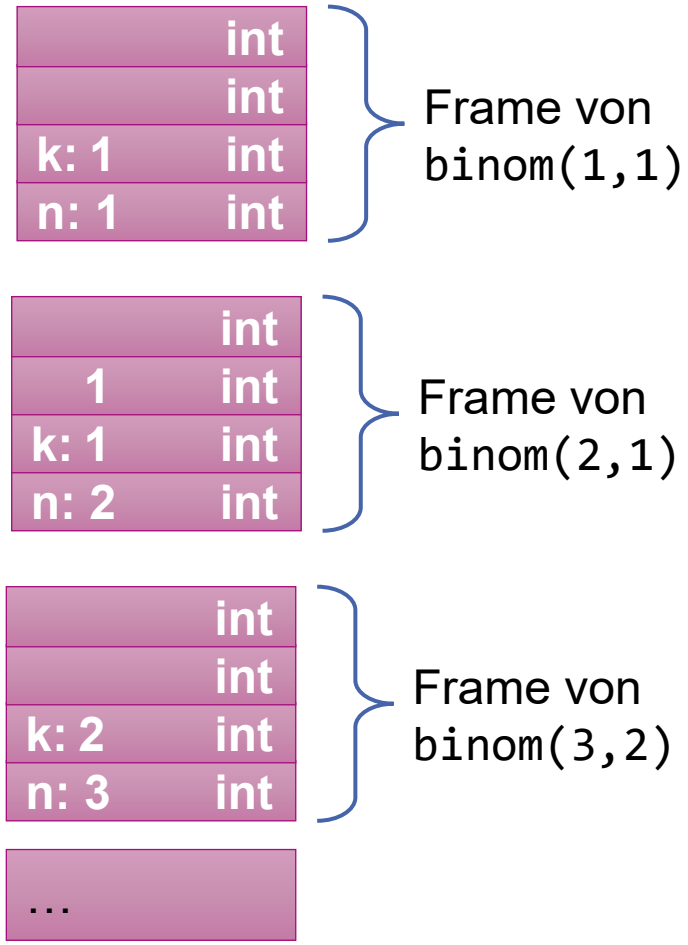


Für jede Methode wird ein neuer Speicherbereich (**Schachtel** bzw. *frame*) auf dem Aufrufstapel angelegt mit u.a.

- Lokalen Variablen (inkl. Parameter)
- Operanden

```
binom(int n, int k) {
  if (k == 0 || k == n) {
      return 1;
    } else {
      return binom(n-1, k-1) + binom(n-1, k);
  }
}
```

Methoden im Aufrufstapel

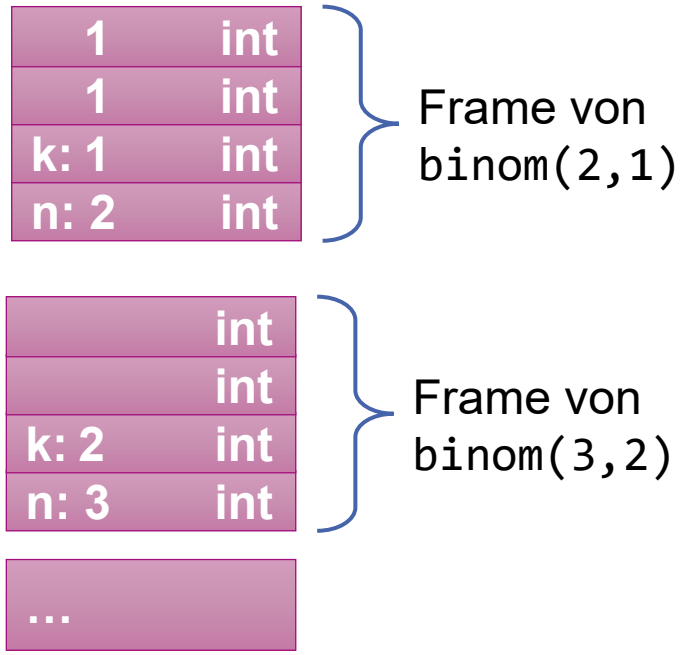


Für jede Methode wird ein neuer Speicherbereich (**Schachtel** bzw. *frame*) auf dem Aufrufstapel angelegt mit u.a.

- Lokalen Variablen (inkl. Parameter)
- Operanden

```
binom(int n, int k) {
  if (k == 0 || k == n) {
      return 1;
    } else {
      return binom(n-1, k-1) + binom(n-1, k);
  }
}
```

Methoden im Aufrufstapel

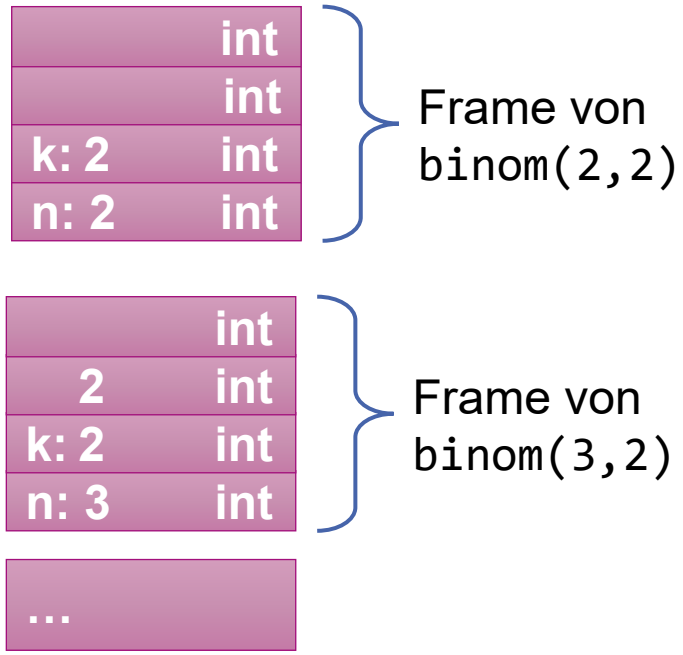


Für jede Methode wird ein neuer Speicherbereich (**Schachtel** bzw. *frame*) auf dem Aufrufstapel angelegt mit u.a.

- Lokalen Variablen (inkl. Parameter)
- Operanden

```
binom(int n, int k) {
  if (k == 0 || k == n) {
      return 1;
    } else {
      return binom(n-1, k-1) + binom(n-1, k);
  }
}
```

Methoden im Aufrufstapel



Für jede Methode wird ein neuer Speicherbereich (**Schachtel** bzw. *frame*) auf dem Aufrufstapel angelegt mit u.a.

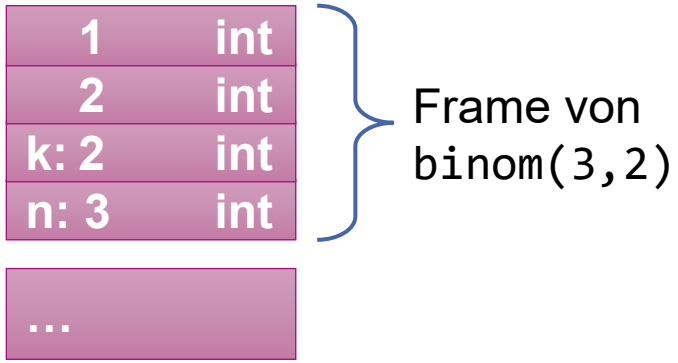
- Lokalen Variablen (inkl. Parameter)
- Operanden

```
binom(int n, int k) {
  if (k == 0 || k == n) {
      return 1;
    } else {
      return binom(n-1, k-1) + binom(n-1, k);
  }
}
```

Methoden im Aufrufstapel

Für jede Methode wird ein neuer Speicherbereich (**Schachtel** bzw. *frame*) auf dem Aufrufstapel angelegt mit u.a.

- Lokalen Variablen (inkl. Parameter)
- Operanden




```
binom(int n, int k) {
  if (k == 0 || k == n) {
      return 1;
    } else {
      return binom(n-1, k-1) + binom(n-1, k);
  }
}
```

Methoden im Aufrufstapel

Für jede Methode wird ein neuer Speicherbereich (**Schachtel** bzw. *frame*) auf dem Aufrufstapel angelegt mit u.a.

- Lokalen Variablen (inkl. Parameter)
- Operanden

→ Wert 3 wird zurückgegeben an aufrufenden Frame

...

```
binom(int n, int k) {
  if (k == 0 || k == n) {
      return 1;
    } else {
      return binom(n-1, k-1) + binom(n-1, k);
  }
}
```

Binomialfunktion mit Caching

Effizientere Implementierung mit Caching:

```
public static int binom(int n, int k) {
  int[][] cache = new int[n+1][k+1];
  return binom(n, k, cache);
}

private static int binom(int n, int k, int[][] cache) {
  if (cache[n][k] == 0) {
    if (k == 0 || k == n) {
      cache[n][k] = 1;
    } else {
      cache[n][k] = binom(n-1, k-1, cache) + binom(n-1, k, cache);
    }
  }
  return cache[n][k];
}
```

Rekursion vs. Iteration

Rekursion

- Vorteilhaft, wenn die Anzahl der Iterationen noch unklar
- Viele Methodenaufrufe
  - ➔ Zeitaufwendiger
- Belegen des Stacks für die Werte der aktuellen und lokalen Variablen
  - ➔ Speicheraufwendiger
- Nicht auf dem ersten Blick abschätzbar, ob sie terminiert.

Iteration

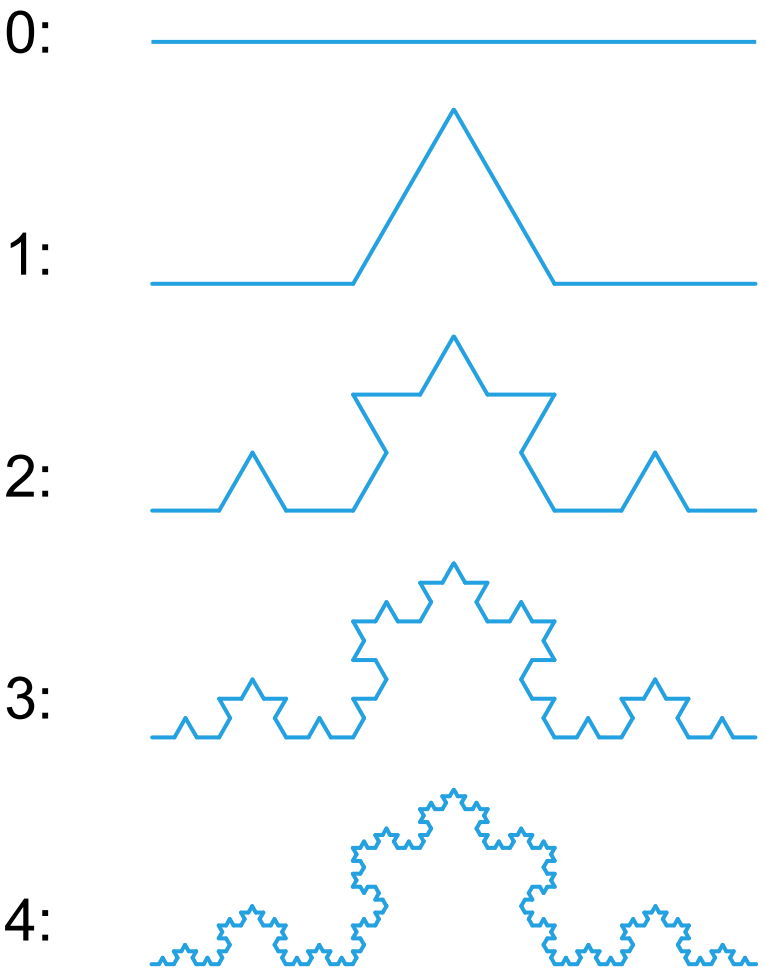
- Vorteilhaft, wenn die Anzahl der Iterationen bekannt ist.
- Komplexität leichter abschätzbar
- Leichter abschätzbar, ob die Iteration terminiert.

Die Kochsche Schneeflockenkurve

Die Koch-Kurve ist ein **Fraktal**.

Sie wird **iterativ** gebildet:

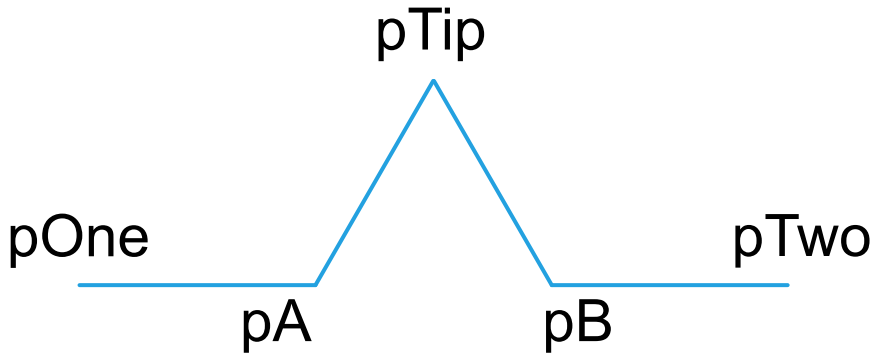
- beginnend mit einer geraden Linie
- aus jedem Geradenstück wird das mittlere Drittel entfernt
- Dafür wird der obere Teil eines gleichseitigen Dreiecks aufgesetzt



Kochsche Schneeflockenkurve: Aufgabe

Implementieren Sie die Schneeflockenkurve rekursiv. Gegeben seien

- Klasse `Point` mit Attributen `int x` und `int y`
- Klasse `Graphics` zum Zeichnen einer Linie:
  - Mit `Graphics g` und `Point p1, p2`:
  - `g.drawLine(p1.x, p1.y, p2.x, p2.y)`
- Parameter `int lev`: Gewünschte Rekursionstiefe
- Hilfsmethoden
  - `getPointA(Point pOne, Point pTwo)`
  - `getPointB(Point pOne, Point pTwo)`
  - `getTip(Point pOne, Point pTwo)`



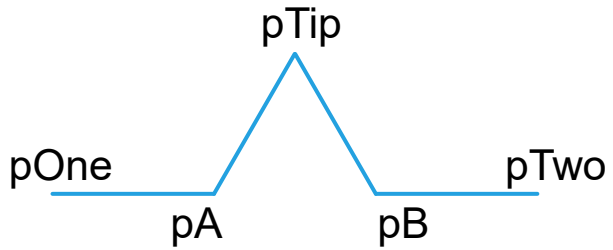
- Zu implementieren:

```
private void drawSegment(Graphics g,
  int lev, Point pOne, Point pTwo){

}
```

Kochsche Schneeflockenkurve

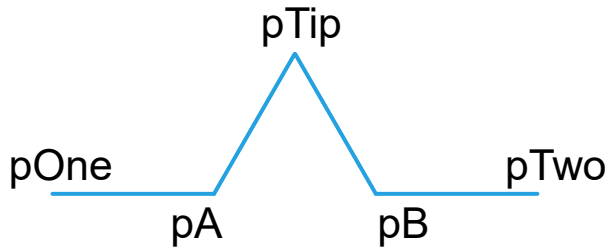
```
private void drawSegment(Graphics g, int lev, Point pOne, Point pTwo) {
  if (lev == 0) {
    // Einfacher Fall
  }
  if (lev >= 1) {
    // Schwieriger Fall
  }
}
```



Kochsche Schneeflockenkurve

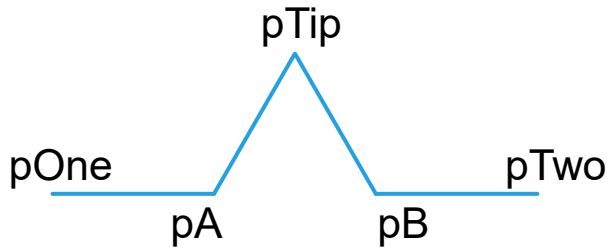
```
private void drawSegment(Graphics g, int lev, Point pOne, Point pTwo) {
  if (lev == 0) {
    g.drawLine(pOne.x, pOne.y, pTwo.x, pTwo.y);
  }
  if (lev >= 1) {
    Point pA = getPointA(pOne, pTwo);
    Point pB = getPointB(pOne, pTwo);
    Point pTip = getTip(pOne, pTwo);
    // wenn lev 1 oder höher,
    // viermal rekursiv sich selbst aufrufen

  }
}
```



Kochsche Schneeflockenkurve

```
private void drawSegment(Graphics g, int lev, Point pOne, Point pTwo) {
  if (lev == 0) {
    g.drawLine(pOne.x, pOne.y, pTwo.x, pTwo.y);
  }
  if (lev >= 1) {
    Point pA = getPointA(pOne, pTwo);
    Point pB = getPointB(pOne, pTwo);
    Point pTip = getTip(pOne, pTwo);
    drawSegment(g, lev - 1, pOne, pA); // wenn lev 1 oder höher,
    drawSegment(g, lev - 1, pA, pTip); // viermal rekursiv
    drawSegment(g, lev - 1, pTip, pB); // sich selbst aufrufen
    drawSegment(g, lev - 1, pB, pTwo);
  }
}
```



Rekursion – Zusammenfassung

- Löse Problem durch Berechnung einer kleineren Instanz des Problems, bis Instanz des Problems so klein ist, dass die Lösung offensichtlich ist.
- Instanzen hängen von Parameter ab.
- Nachweis der Termination notwendig.
- Speicherverbrauch linear zur Zahl der Aufrufe (für lokale Variablen, Rücksprung-Adresse und aktuelle Parameter, für die in jeder Instanz neuer Speicher belegt wird).

Literaturhinweis – Weiterlesen

Dietmar Ratz, Jens Scheffler, Detlef Seese und Jan Wiesenberger
Grundkurs Programmieren in Java, 7. Auflage, 2014 (mit Java 8), Hansa-Verlag

- *Methoden, Unterprogramme*
  - *Rekursiv definierte Methoden*