Prozeduren in Java

- Die Erweiterung des Modulbegriffs um Prozeduren ermöglicht uns nun natürlich auch, diese Prozeduren in anderen Algorithmen zu verwenden
- Das Modul `PhysikImperativ` stellt z.B. die Methode (Prozedur `streckeImperativ` zur Verfügung
- D.h. wir können nun alle Methoden, die wir in irgendwelchen Modulen implementiert haben wie Basisoperationen handhaben und *aufrufen*
- Solch ein Methodenaufruf ist syntaxtisch ein Ausdruck, kann also überall stehen, wo ein Ausdruck stehen darf (die formale Erweiterung der induktiven Definition von Ausdrücken ist trivial, und sei dem Leser überlassen)

Prozeduren in Java

- Eine Methode mit Ergebnistyp `<typ>` $\neq$ **`void`** ist ein Ausdruck vom Typ `<typ>` und kann überall dort stehen, wo ein Ausdruck vom Typ `<typ>` verlangt ist (z.B. bei einer Wertzuweisung an eine Variable vom Typ `<typ>`)
  Beispiel: `double s = strecke(3.0, 4.2, 7.1);`
  Achtung: da Java nicht zwischen Funktion und Prozedur unterscheidet, könnte die Methode `strecke` Nebeneffekte haben!
- Eine Methode mit Ergebnistyp **`void`** ist ein Ausdruck vom Typ **`void`** und kann als Ausdrucksanweisung verwendet werden; die Seiteneffekte der Methode ist das (hoffentlich) beabsichtigte Resultat der Anweisung.
  Beispiel: `System.out.println` ist eine Methode mit Ergebnistyp **`void`** und hat als Seiteneffekt: gib den Eingabeparameter (Typ `String` für Zeichenketten) auf dem Bildschirm aus, d.h.
  `System.out.println(...);` ist eine Ausdrucksanweisung

Prozeduren in Java

- Genauso wie bei Funktionen stellt sich auch für die Prozeduren die Frage: was passiert, wenn eine Prozedur ausgeführt wird, also die Anweisungen im Rumpf der Prozedur ausgeführt werden?
- Wir klären diese Frage zunächst wieder informell mit unserer Intuition, dass eine Variable/Konstante ein Zettel ist
- Offenbar bewirkt eine Anweisung eine *Veränderung des Zustands* indem sich ein Programm (in Ausführung) gerade befindet
- Eine Wertzuweisung an eine Variable z.B. verändert den Zustand so, dass die Variable einen neuen Wert bekommt (also ab jetzt neu substituiert wird)
- Stellt sich noch die Frage, wie so ein Zustand aussehen kann: intuitiv besteht ein Zustand aus der Menge an Zetteln, die momentan vereinbart sind

Prozeduren in Java

- Formal ist ein *Zustand* (engl. *State*) eine Menge $\mathcal{S}$ von Paaren $z = (x, d)$ mit folgenden Eigenschaften
  - x ist ein Eingabeparameter, eine (vereinbarte) Konstante oder eine (vereinbarte) Variable und heißt *Name* oder *Bezeichner* von z
  - d ist ein Objekt der Sorte von x (oder *undefiniert*, notiert als ω) und heißt *Inhalt* von z
  - Die Menge $\mathcal{S}$ enthält keine zwei verschiedenen Paare (x, d_1) und (x, d_2) mit gleichem Namen x
- Ein Paar (x, d) formalisiert das Bild eines Zettels x auf dem der Wert d steht
- (x, ω) kann als *leerer Zettel* verstanden werden

Prozeduren in Java

- Sei $N(\mathcal{S})$ die Menge aller Namen (Variablen und Konstanten) in $\mathcal{S}$ und $V(\mathcal{S}) \subseteq N(\mathcal{S})$ die Menge der (formalen) Eingabeparameter der aufgerufenen Prozedur
- Ein Zustand $\mathcal{S}$ definiert eine Substitution für alle Namen $x \in N(\mathcal{S})$ in $\mathcal{S}$ deren Wert $d \neq \omega$ ist:
  das Paar $(x, d) \in \mathcal{S}$ definiert offenbar die Substitution $\sigma = [x/d]$
- Dies kommt Ihnen bekannt vor:
  - Bei Funktionen haben wir gesehen, dass der Aufruf eines Algorithmus eine Substitution σ für die Eingabeparameter des Algorithmus $V(\sigma)$ spezifiziert (analog gilt dies nun für $V(\mathcal{S})$, das wir bisher mit $V(\sigma)$ bezeichnet hatten).
  - Zustände beinhaltet offenbar nun aber nicht mehr nur die Eingabeparameter sondern auch die im Rumpf zusätzlich vereinbarten Konstanten und Variablen, daher ist $N(\mathcal{S})$ auch wirklich eine Obermenge von $V(\mathcal{S})$

Prozeduren in Java

- Dieser Unterschied zwischen Funktion und Prozedur drückt sich auch in unserer unterschiedlichen Schreibweise aus: σ bzw. $\mathcal{S}$ und V bzw. N
- Offensichtlich ist $(e, d) \in \mathcal{S}$ mit $d \neq \omega$ für alle *Eingabeparameter* $e \in V(\mathcal{S})$, d.h. alle Variablen, die Eingabeparameter bezeichnen, sind definiert/nicht leer (der Aufruf der Prozedur mit konkreten Eingabewerten erwirkt eine entsprechende Substitution); $V(\mathcal{S})$ bezeichnet also unser bisheriges $V(\sigma)$
- Dies gilt aber nicht notwendigerweise für alle vereinbarten *lokalen Variablen und Konstanten* $N(\mathcal{S}) \setminus V(\mathcal{S})$, denn es können Variablen zunächst nur vereinbart werden (ohne, dass sie sofort initialisiert werden)

Prozeduren in Java

- Damit wird der Wert eines Ausdrucks u nun abhängig vom Zustand $\mathcal{S}$, in dem er betrachtet wird; wir schreiben daher nun $W_{\mathcal{S}}$ statt W_σ (da $\mathcal{S}$ ja auch eine Substitution definiert)
- Die formale Definition von $W_{\mathcal{S}}$ ist allerdings leicht: es genügt eine Erweiterung der rekursiven Definition des Wertes eines Ausdrucks u für den Fall, dass u eine lokale Variable oder Konstante aus $N(\mathcal{S}) \setminus V(\mathcal{S})$ ist
- Die anderen Fälle, dass u ein Eingabeparameter, ein Literal, die Anwendung eines Grund-Operators oder ein Aufruf einer Funktion (mit Rückgabewert ohne Seiteneffekte) ist, bleiben unverändert (d.h. in diesen Fällen ersetzen wir $W_\sigma(u)$ durch $W_{\mathcal{S}}(u)$)
- Für den Fall $u = x$ mit $x \in N(\mathcal{S}) \setminus V(\mathcal{S})$ (lokale Variable/Konstante) definieren wir nun zusätzlich
  - ist $(x, d) \in \mathcal{S}$ und $d \neq \omega$, so ist $W_{\mathcal{S}}(u) = d$
  - ist $(x, \omega) \in \mathcal{S}$ oder $x \notin N(\mathcal{S})$, so ist $W_{\mathcal{S}}(u) = \omega$
- Wir schreiben $W(u)$ statt $W_{\mathcal{S}}(u)$, wenn $\mathcal{S}$ im Kontext klar ist

Prozeduren in Java

- Was passiert mit einem Zustand $\mathcal{S}$, wenn Anweisungen ausgeführt werden, die Variablen aus $N(\mathcal{S})$ verändern bzw. neue vereinbaren?
- Intuitiv:
  - bei der Vereinbarung einer Variablen (analog Konstanten) wird ein neuer Zettel hinzugefügt, der Zettel ist zunächst leer
  - bei der Initialisierung einer Variablen (analog Konstanten) wird der entsprechende Zettel (erstmalig) verändert (bildlich: der leere Zettel wird durch einen neuen, nun nicht-leeren Zettel gleichen Namens ersetzt)
  - bei einer weiteren Wertzuweisung an eine Variable wird der entsprechende Zettel (wieder) verändert

Prozeduren in Java

- Formal: sei `a` eine Variablen- bzw. Konstantenvereinbarung, eine Initialisierung oder eine Wertzuweisung
- `a` bewirkt eine *Zustandsänderung* von Zustand $\mathcal{S}$ in einen *Nachfolgezustand* $\hat{\mathcal{S}}$ wie folgt
  - Vereinbarung:
    Hat `a` die Form `<type> x;` mit $\mathtt{x} \notin N(\mathcal{S})$, so ist $\hat{\mathcal{S}} = \mathcal{S} \cup \{(\mathtt{x}, \omega)\}$
  - Initialisierung:
    Hat `a` die Form `x = u;` mit $\mathtt{x} \in N(\mathcal{S})$ und ist `u` ein Ausdruck mit $W(\mathtt{u}) \neq \omega$, so ist $\hat{\mathcal{S}} = \mathcal{S} \setminus \{(\mathtt{x}, \omega)\} \cup \{(\mathtt{x}, W(\mathtt{u}))\}$
  - Wertzuweisung:
    Hat `a` die Form `x = u;` mit $(\mathtt{x}, d) \in \mathcal{S}$ (d.h. $\mathtt{x} \in N(\mathcal{S})$ hat den Wert d) und ist `u` ein Ausdruck mit $W(\mathtt{u}) \neq \omega$, so ist $\hat{\mathcal{S}} = \mathcal{S} \setminus \{(\mathtt{x}, d)\} \cup \{(\mathtt{x}, W(\mathtt{u}))\}$

Prozeduren in Java

- Wie bereits erwähnt enthält der Anfangszustand beim Aufruf eines Algoroithmus (einer Prozedur) $\mathcal{S}_0$ bereits die Eingabevariablen $V(\mathcal{S})$ des Algorithmus, der ausgeführt wird mit den entsprechenden konkreten Eingabe-Werten des Aufrufs
- Wir schreiben $\mathcal{S} \xrightarrow{\texttt{a}} \hat{\mathcal{S}}$, um auszudrücken, dass wir $\hat{\mathcal{S}}$ aus $\mathcal{S}$ durch die Anwendung von `a` erhalten haben
- Ist `A` eine Folge von Variablen-/Konstantenvereinbarungen, Initialisierungen und/oder Wertzuweisungen `a1; ... ak;` und es gilt $\mathcal{S} = \mathcal{S}_0 \xrightarrow{\texttt{a1}} \ldots \xrightarrow{\texttt{ak}} \mathcal{S}_k = \hat{\mathcal{S}}$, so heißt $\hat{\mathcal{S}}$ Nachfolgezustand von $\mathcal{S}$ bzgl. `A` und wir schreiben $\mathcal{S} \xrightarrow{\texttt{A}} \hat{\mathcal{S}}$

Prozeduren in Java

- Beispiel:

```
public static void beispielMethode1(int a) {
    int x;
    int y = 5;
    x = 0;
    final int z = 3;
    x += 7 + y;
    y = y + z - 2;
    x = x * y;
    x = x/z;
    x += a;
    a = x + y - z;
}
```

Prozeduren in Java

- Ausführung des Aufrufs `beispielMethode1(1)`:

Anweisung	x	y	z	a	Zustand
beim Aufruf	■	■	■	1	$\mathcal{S}_0 = \{(a,1)\}$
`int x;`	ω	■	■	1	$\mathcal{S}_1 = \{(a,1),(x,\omega)\}$
`int y = 5;`	ω	5	■	1	$\mathcal{S}_2 = \{(a,1),(x,\omega),(y,5)\}$
`x = 0;`	0	5	■	1	$\mathcal{S}_3 = \{(a,1),(x,0),(y,5)\}$
`final int z = 3;`	0	5	$\|3\|$	1	$\mathcal{S}_4 = \{(a,1),(x,0),(y,5),(z,3)\}$
`x += 7 + y;`	12	5	$\|3\|$	1	$\mathcal{S}_5 = \{(a,1),(x,12),(y,5),(z,3)\}$
`y = y + z - 2;`	12	6	$\|3\|$	1	$\mathcal{S}_6 = \{(a,1),(x,12),(y,6),(z,3)\}$
`x = x * y;`	72	6	$\|3\|$	1	$\mathcal{S}_7 = \{(a,1),(x,72),(y,6),(z,3)\}$
`x = x /z;`	24	6	$\|3\|$	1	$\mathcal{S}_8 = \{(a,1),(x,24),(y,6),(z,3)\}$
`x += a;`	25	6	$\|3\|$	1	$\mathcal{S}_9 = \{(a,1),(x,25),(y,6),(z,3)\}$
`a = x + y - z;`	25	6	$\|3\|$	28	$\mathcal{S}_{10} = \{(a,28),(x,25),(y,6),(z,3)\}$

Legende:

■ = Zettel existiert noch nicht $\quad\quad$ $\|\ldots\|$ = Zettel nicht radierbar (Konstante)

- Es gilt: $\mathcal{S}_0 \xrightarrow{\mathtt{A}} \mathcal{S}_{10}$, wobei `A` den Rumpf der Methode bezeichnet

Prozeduren in Java

- Die nächste Frage, die wir uns gleich im Anschluss stellen sollten ist: was passiert, wenn ein Algorithmus eine andere Prozedur aufruft (ein solcher Aufruf ist ja eine erlaubte Anweisung; die konkrete Eingabe kann syntaktisch sogar ein beliebiger Ausdruck des entspr. Typs sein)?
- Beispiel (wieder sinnfrei):

```
public static int beispielMethode2(int a) {
   int x = a;
   int y = a;
   beispielMethode1(a);
   return x + y + a;
}
```

- Bei Aufruf `beispielMethode2(1)`
  1. Was für einen Effekt hat der Aufruf von `beispielMethode1` auf `a` (ein Nebeneffekt von `beispielMethode1` ist ja die Veränderung des Wertes der Eingabe-Variable `a`)?
  2. Welche Variablen `x` und `y` sind wann sichtbar/gültig (beide Methoden benutzen Variablen mit den Namen `x` und `y`)?

Prozeduren in Java

- Zur ersten Frage: Wir klären zunächst ganz allgemein, was passiert, wenn eine Methode (Prozedur) mit Signatur

  ```
  <outType> p(<inType1> e1, ... , <inTypeN> eN)
  ```

  innerhalb des Rumpfes einer anderen Methode `m` mit Variablen `x1, ... , xN` vom Typ `<inType1>, ... , <typeN>`, die in `m` bekannt sind, als Argument aufgerufen wird, d.h. im Rumpf von `m` steht die Anweisung

  ```
  ... p(x1, ... , xN);
  ```

  Wir sagen hier auch, die Variablen `x1, ... , xN` werden an `p` *übergeben*
- Variablen können grundsätzliche auf zwei Arten übergeben werden

Prozeduren in Java

- Möglichkeit 1: *Call-by-value*
  - Für jede (formale) Eingabe-Variable `e1, ... , eN` (der Signatur) wird im Methoden-Block eine neue Variable (ein Extra-Zettel) angelegt
  - Diese Extra-Variablen erhalten die *Werte* der übergebenen Variablen `x1, ... , xN` (bzgl. des Zustands zum Zeitpunkt des Aufrufs von `p` in `m`)
  - Die übergebenen Variablen `x1, ... , xN` sind im Rumpf von `p` nicht sichtbar (siehe auch Frage 2 von oben)
  - In unserer Beispiel-Methode `beispielMethode1` wird für `a` im Block eine eigene Variable `aNeu` angelegt und mit dem Wert des übergebenen Ausdrucks (hier der aktuelle Wert von `a` in `beispielMethode2`, der zum Zeitpunkt des Aurufs `1` ist) belegt; im Rumpf von `beispielMethode2` ist das ursprünglich `a` also auch nicht sichtbar, sondern nur `aNeu`
  - Auf diese Weise bleibt der Wert der ursprüngliche Variable von Anweisungen innerhalb der Methode unberührt

Prozeduren in Java

- Möglichkeit 2: *Call-by-reference*
  - Für jede (formale) Eingabe-Variable `e1, ... , eN` (der Signatur) wird im Methoden-Block *keine* neue Variable angelegt
  - Die Eingabe-Variable `e1, ... , eN` erhalten stattdessen eine *Referenz* (*Verweis*) auf die übergebenen Variablen `x1, ... , xN`
  - Wenn innerhalb der Methode der Wert einer der Eingabe-Variablen `e1, ... , eN` verändert wird, wird also in Wirklichkeit eine der übergebenen Variablen `x1, ... , xN` verändert, obwohl diese im Rumpf von `p` gar nicht sichtbar sind
  - Das hat dann offenbar auch Auswirkungen außerhalb der Methode
- *Achtung: call-by-reference ist daher eine potentielle Quelle unbeabsichtigter Seiteneffekte!!!*

Prozeduren in Java

- Zur zweiten Frage: der Rumpf einer Methode definiert einen Anweisungsblock
- Es gelten die bereits bekannten Regeln zu Lebensdauer, Sichtbarleit, etc. für Blöcke
- Zusätzlich zu den lokalen Variablen, die innerhalb des Blocks vereinbart werden, sind noch die formalen Eingabe-Variablen der Methode sichtbar (so sind wir es ja auch gewohnt)
- In unserem konkreten Beispiel würde beim Aufruf von `beispielMethode1` innerhalb des Rumpfes von `beispielMethode1` neue Variablen `xNeu` und `yNeu` angelegt werden, die unabhängig von den Variablen `x` und `y` sind
- Aus dem selben Grund sind die übergebenen Variablen (in unserem Beispiel `a`) im Rumpf nicht sichtbar

Prozeduren in Java

- Beispiel: Aufruf von `beispielMethode2(1)` mit call-by-value

```
1  public static int beispielMethode2(int a) {
2      int x = a;
3      int y = a;
4      beispielMethode1(a);
5      return x + y + a;
6  }
```

  - Zustand nach Zeile 3 ist $\mathcal{S}_3 = \{(\mathtt{a}, 1), (\mathtt{x}, 1), (\mathtt{y}, 1)\}$
  - Beim Aufruf von `beispielMethode1(a)` in Zeile 4 wird ein *zweiter* Zettel mit Namen `a` (zur Unterscheidung `aNeu` genannt) mit $(\mathtt{aNeu}, 1)$ angelegt, der nur im Rumpf von `beispielMethode1` gilt
  - Innerhalb des Rumpfes von `beispielMethode1` werden neue Zettel für `x` und `y` (sowie `z`) angelegt, die unabhängig von `x` und `y` sind
  - Nach Abarbeitung von `beispielMethode1` in Zeile 4 ist daher wieder der Zustand $\mathcal{S}_4 = \{(\mathtt{a}, 1), (\mathtt{x}, 1), (\mathtt{y}, 1)\}$, da die (ursprünglichen) Zettel `a`, `x` und `y` *nicht* verändert wurden
  - Rückgabewert von `beispielMethode2` ist demnach 1+1+1 = 3

Prozeduren in Java

- Beispiel: Aufruf von `beispielMethode2(1)` mit call-by-reference

```
1  public static int beispielMethode2(int a) {
2      int x = a;
3      int y = a;
4      beispielMethode1(a);
5      return x + y + a;
6  }
```

  - Zustand nach Zeile 3 ist $\mathcal{S}_3 = \{(\mathtt{a}, 1), (\mathtt{x}, 1), (\mathtt{y}, 1)\}$
  - Beim Aufruf von `beispielMethode1(a)` in Zeile 4 wird *kein* neuer Zettel mit Namen `a` angelegt sondern der existierende Zettel $(\mathtt{a}, 1)$ im Rumpf von `beispielMethode1` verwendet
  - Variablen `x` und `y` werden natürlich nach wie vor nicht verändert
  - Nach Abarbeitung von `beispielMethode1` in Zeile 4 ist der Zettel `a` nun durch `beispielMethode1` verändert worden (Aufruf von `beispielMethode1(1)` schreibt am Ende den Wert 28 auf den Zettel `a`)
  - Zustand nach Zeile 4 ist daher $\mathcal{S}_4 = \{(\mathtt{a}, 28), (\mathtt{x}, 1), (\mathtt{y}, 1)\}$
  - Rückgabewert von `beispielMethode2` ist demnach 1+1+28 = 30

Prozeduren in Java

- Java wertet Parameter call-by-value aus
- Dies mach insbesondere deshalb Sinn, weil in Java nicht nur Variablen sondern auch ganze Ausdrücke an Methoden übergeben werden dürfen (so hatten wir übrigens auch die Syntax von Ausdrücken definiert), als z.B. ist

```
beispielMethode1(a+1);
```

  ein erlaubter Aufruf, übergeben wird der (Wert des) Ausdruck(s) `a+1`
- Der Wert von Ausdrücken und Zustandsübergängen bei Methodenaufrufen kann als call-by-value Umsetzung formal definiert werden
- Wir erweitern dazu die Formalisierung der Zustandsübergänge, die eine Anweisung `a` bewirkt um folgende Fälle

Prozeduren in Java

- Beim Aufruf einer Prozedur **void** `p(<type1> e1, ... ,<typeN> eN)` mit Eingabeparametern `e1`, ... `eN` und konkreten Eingaben (Ausdrücke) `w1`, ... `wN` (d.h. Anweisung `a` hat die Form `p(w1, ..., wN);`) im Zustand $\mathcal{S}$ wird zunächst der Zustand

$$\mathcal{S}_{init} = \{(\texttt{e1}, W_{\mathcal{S}}(\texttt{w1})), \ldots, (\texttt{eN}, W_{\mathcal{S}}(\texttt{wN}))\}$$

  erreicht, d.h., $\mathcal{S}_{init}$ enthält nur die Eingabe-Varaiblen mit den Werten der übergebenen Ausdrücke bzgl. $\mathcal{S}$
- Der Rumpf `r` von `p` überführt $\mathcal{S}_{init}$ in $\mathcal{S}_{end}$, d.h. $\mathcal{S}_{init} \xrightarrow{\texttt{r}} \mathcal{S}_{end}$
- Da `p` keinen Rückgabewert hat (Typ **void**), ist der Nachfolgezustand $\hat{\mathcal{S}} = \mathcal{S}$, d.h. $\mathcal{S} \xrightarrow{\texttt{a}} \mathcal{S}$ (die gewünschten Seiteneffekte haben also keinen Einfluss auf $\hat{\mathcal{S}}$ bzw. $\mathcal{S}$)

Prozeduren in Java

- Beim Aufruf einer Prozedur
  `<type0> p(<type1> e1, ... ,<typeN> eN)` mit Eingabeparametern `e1`, ... `eN` und konkreten Eingaben (Ausdrücke) `w1`, ... `wN` in einer Zuweisung an eine Variable `x` (d.h. Anweisung `a` hat die Form `x = p(w1, ..., wN);`) im Zustand $\mathcal{S} = \{\ldots(\mathtt{x},\mathtt{w})\ldots\}$ (wobei $\mathtt{w} = \omega$ sein kann) wird wiederum zunächst der Zustand

  $$\mathcal{S}_{init} = \{(\mathtt{e1}, W_{\mathcal{S}}(\mathtt{w1})), \ldots, (\mathtt{eN}, W_{\mathcal{S}}(\mathtt{wN}))\}$$

  erreicht, d.h., $\mathcal{S}_{init}$ enthält nur die Eingabe-Varaiblen mit den Werten der übergebenen Ausdrücke bzgl. $\mathcal{S}$
- Der Rumpf `r` von `p` überführt $\mathcal{S}_{init}$ in $\mathcal{S}_{end}$, d.h. $\mathcal{S}_{init} \xrightarrow{\mathtt{r}} \mathcal{S}_{end}$
- Die **`return`**-Anweisung von `p` enthält den Ausdruck `t` vom Typ `<type0>` mit Wert $W_{\mathcal{S}_{end}}(\mathtt{u})$ im Zustand $\mathcal{S}_{end}$
- Der Nachfolgezustand ist damit gegeben durch $\hat{\mathcal{S}} = \mathcal{S} \setminus \{(\mathtt{x},\mathtt{w})\} \cup \{(\mathtt{x}, W_{\mathcal{S}_{end}}(\mathtt{u}))\}$ und es gilt wieder $\mathcal{S} \xrightarrow{\mathtt{a}} \hat{\mathcal{S}}$

Prozeduren in Java

- Noch ein Beispiel

```
public class Exchange
{
      public static void swap(int i, int j) {
            int c = i;
            i = j;
            j = c;
      }

      public static void main(String[] args) {
            int x = 1;
            int y = 2;
            swap(x,y);
            System.out.println(x); // Ausgabe?
            System.out.println(y); // Ausgabe?
      }
}
```

Prozeduren in Java

- Aufruf von `main`:
  - Zunächst werden nacheinander $(\texttt{x}, 1), (\texttt{y}, 2)$ angelegt (wir ignorieren `args` im Kopf von `main`)
  - Beim Aufruf von `swap` werden neue Zettel angelegt: `i` als Kopie für `x` und `j` als Kopie für `y`:
    $(\texttt{i}, 1), (\texttt{j}, 2)$
  - Dann
    $(\texttt{i}, 1), (\texttt{j}, 2), (\texttt{c}, 1)$
    $(\texttt{i}, 2), (\texttt{j}, 2), (\texttt{c}, 1)$
    $(\texttt{i}, 2), (\texttt{j}, 1), (\texttt{c}, 1)$
  - Alles schön und gut, aber nach dem Aufruf von `swap` in `main` (beim Ausgabe mit `System.out`) hat `x` den Wert 1 und `y` den Wert 2, d.h. `swap` hat keinerlei Wirkung auf `x` und `y` in `main`

Und das ist genau das, was wir formalisiert haben!

Prozeduren vs. Funktionen

- Mit der Formalisierung wird auch der Unterschied zwischen Prozeduren und Funktionen noch einmal klar
- Der Aufruf beider Varianten bewirkt zunächst das Gleiche: die Eingabevariablen werden mit konkreten Werten belegt
- Bei Funktionen wird anschließend der Wert des Rumpfes der Funktion (ein Ausdruck) ausgewertet; der Funktionsaufruf ist ja selbst wieder ein Ausdruck (mit einem Wert)
- Dabei wird die Variablenbelegung innerhalb des Rumpfes der Funktion nicht verändert, wir haben keine Zustandsänderung (daher hatten wir auch keinen Zustandsbegriff benötigt)

Prozeduren vs. Funktionen

- Bei Prozeduren können weitere Variablen und Konstanten (mit entsprechenden Belegungen) zu der Menge der Eingabevariablen hinzukommen
- Die Anweisungen im Rumpf verändern ggf. deren Belegungen (das sind die vielbeschworenen Seiteneffekte)
- Der Wert von Ausdrücken ist damit abhängig von der aktuellen Belegung; daher hatten wir den Zustandsbegriff eingeführt
- Bei der (nicht seltenen) Mischform, Prozeduren die einen Rückgabewert vom Typ $\mathtt{T} \neq \mathtt{void}$ besitzen, stellt der Ausdruck nach der `return`-Anweisung also nicht notwendigerweise den direkten Zusammenhang zwischen Eingabewerten und Ausgabe dar, da der Zustand am Ende des Methodenrumpfes typischerweise nicht mehr mit dem Startzustand (bei Aufruf) übereinstimmt
- Das erklärt auch die Umsetzung der Funktion in Java als Methode in der nur `return`-Anweisungen vorkommen