

Java für Scheme-Könner

Dieses Dokument ist ein Kurzüberblick über die wichtigsten Sprachelemente von Java, jeweils im Vergleich mit den bekannten Mechanismen von Scheme. Die Erklärung ist möglichst knapp gehalten. Java ist eine sehr große Sprache; viele Details sind darum hier nicht beschrieben. Im wesentlichen soll das Dokument als „Spickzettel“ für die erste Java-Runde dienen. Auf Reihenfolge in der Darstellung wurde dabei nur begrenzt Rücksicht genommen.

1 Kurzüberblick

Statische Typen Scheme besitzt ein sogenanntes *latentes Typsystem*: jeder Wert in einem Scheme-Programm trägt Typinformation mit sich, die durch entsprechende Prädikate (`null?`, `pair?`, `number?`) abgefragt werden kann. Damit existieren die Typinformationen zur Laufzeit und heißen damit gelegentlich auch *dynamische Typen*.

In Java haben auch die Variablen Typen, die bei der Bindung (in Java genannt *Variablendeklaration*) mit angegeben werden. Da bereits vor Programmablauf die meisten Typen feststehen, heißt ist das Java-Typsystem ein *statisches Typsystem*.

Als Typen dienen dabei die sogenannten *primitiven Typen* für Booleans, Zahlen, Buchstaben und Fließkommazahlen:

```
boolean x = true;
int a = 5;
char c = 'b';
double pi = 3.14159265;
```

Als Typen können des weiteren auch die Namen von Klassen dienen:

```
java.util.Date myBirthDate =
    new java.util.Date(71, 4, 11, 22, 00);
```

Diese Typen heißen auch *Referenztypen*, weil ihre Werte tatsächlich Zeiger auf Datenstrukturen sind. Dies entspricht genau der Repräsentation von Datenstrukturen in Scheme.

Referenztypen und primitive Typen Werte von primitiven Typen und Objekte sind in Java nicht vertauschbar: Primitive Werte gehören keiner Klasse an. Also:

```
Object s = "Bla"; // OK
Object n = 27; // verboten
```

`Object` ist eine eingebaute Klasse, die an der Spitze der Klassenhierarchie steht: Jede Klasse ist eine Unterklasse von `Object`. Darum kann `Object` als Typ auch *alle* Werte von Referenztyp aufnehmen.

Typen werden auch für die Rückgabewerte von Methoden angegeben werden. Dort kann auch der Pseudo-Typ `void` stehen, was für „kein Rückgabewert“ steht:

```
public void killMike();
```

Type-Casts Oft weiß die Programmiererin besser als der Java-Compiler, welchen Typ ein Ausdruck hat. Bei Referenztypen weiß z.B. der Java-Compiler oft nur, daß es sich um `Object` handelt. Zum Beispiel verlieren Objekte auf dem Weg in und aus einer Datenstruktur ihren spezifischen Typ

```

Stack s = new Stack();

s.push("Mike ist doof.");
// s.pop() hat den Typ Object!

(s.pop()).length() // verboten:
                    // length ist Methode von String,
                    // aber nicht von Object.
                    // s.pop() hat Typ Object.

```

In diesem Fall kann ein *Type-Cast* — ein Typ in Klammern vor einem Ausdruck — im Programm die fehlende Information ergänzen:

```
((String) s.pop()).length() // OK
```

Variablen In Java können Variablen an zwei Stellen auftauchen:

- *Instanzvariablen* stehen im Rumpf einer Klassendefinition, aber außerhalb eines Methodenrumpfes. Diese Variablen sind in allen Methoden der Klasse sichtbar.
- *Lokale Variablen* stehen innerhalb von Methodenrumpfen, in der Regel am Anfang. (Allerdings kann dort prinzipiell auch geschachtelt werden). Diese Variablen sind jeweils in der Methode sichtbar, in dem sie deklariert werden.

Namenskonventionen Während in Scheme Bezeichner per Konvention in der Regel klein geschrieben und Komposita durch Bindestrich getrennt werden, werden in Java Komposita zusammengeschrieben, wobei der Anfang neuer Wortbestandteile durch einen Großbuchstaben markiert wird: Aus `kill-mike` wird `killMike`. Per Konvention werden die (ersten Buchstaben von) Namen von Klassen großgeschrieben, Variablen klein. Java unterscheidet — im Gegensatz zu Scheme — zwischen Groß- und Kleinschreibung.

Infix-Schreibweise Für arithmetische und einige andere Operatoren wird in Java die gewohnt Infixschreibweise benutzt:

```

15 + 23
a < b
c >= 0

```

Java erzwingt dabei nicht vollständige Klammerung. In Zweifelsfällen bezüglich des Operatorvorrangs empfiehlt sich allerdings, lieber ein Klammernpaar zuviel als eines zuwenig zu benutzen.

Zuweisungen vs. Vergleiche Verwirrenderweise ist der Operator für Vergleiche (also das Pendant zu `eq?`) `==`. `=` ist für die Zuweisung zuständig. Beide werden häufig verwechselt.

```

5 == 7 // OK
5 = 7  // verboten

boolean a = false;
boolean b = (a = true); // OK, aber möglicherweise unerwartet

```

Ausdrücke vs. Befehle und Semikolons Bei den fundamentalen Sprachbestandteilen in Java wird zwischen Ausdrücken und Befehlen unterschieden: Ausdrücke haben immer einen Wert, Befehle niemals. (In Scheme hat jeder Ausdruck einen Wert, auch wenn dieser manchmal unsichtbar ist, wie z.B. bei `set!`.) Sowohl Ausdrücke als auch Befehle können Seiteneffekte haben. Ein Ausdruck kann stets dort stehen, wo ein Befehl steht, aber nicht umgekehrt:

```
x + 5; // Ausdruck als Befehl, OK

// Befehl, OK
if (x < 5) {
    a = 1;
} else {
    a = 2;
}

// Befehl als Ausdruck, verboten
a = if (x < 5) { 1; } else { 2; };

// Wert-Conditional, OK
a = (x < 5) ? 1 : 2;
```

Befehle werden grundsätzlich durch ein Semikolon abgeschlossen.

Wertrückgabe mit return Der Befehl

```
return e;
```

führt dazu, daß sofort der Wert von *e* zurückgegeben wird — die Continuation der Methode wird direkt aufgerufen:

```
class A {
    int x = 5;

    public int m() {

        return 17;

        x = 23; // wird nie ausgeführt
    }
}
```

Endrekursion und Schleifen Endrekursive Methodenaufrufe gibt es in Java nicht. Endrekursion muß somit über spezielle Sprachkonstrukte für Schleifen abgewickelt werden.

```
while (w) {
    b
}
```

führt den Rumpf *b* wiederholt aus, bis die Bedingung *w* falsch wird. Die Bedingung wird jeweils *vor* jedem Schleifendurchlauf getestet. Beispiel:

```
int i = 0;
while (i < 5) {
```

```

    System.out.println("Mike ist doof.");
    i = i + 1;
}

```

Bei der `for`-Schleife

```

for (i; w; u) {
    b
}

```

wird zuerst der Initialisierungsausdruck *i* ausgewertet. Danach wird der Rumpf *b* so oft ausgewertet, bis die Bedingung *w* falsch wird. Die Bedingung wird jeweils *vor* jedem Schleifendurchlauf getestet. Nach jedem Schleifendurchlauf wird *u* ausgewertet. Beispiel:

```

int i;
for (i = 0; i < 5; i = i+1) {
    System.out.println("Mike ist doof");
}

```

Sichtbarkeitsdeklarationen Bei Methoden und Instanzvariablen lässt sich bei der Deklaration kontrollieren, ob diese von außen im gleichem Package, in Unterklassen etc. sichtbar sind. Die kompletten Regeln hierfür sind sehr kompliziert. Als Faustregel genügt es meist, nur die Sichtbarkeiten `private` und `public` zu verwenden: `public` ist überall, `private` nur innerhalb der Klasse sichtbar. Grundsätzlich empfiehlt sich sogar, Instanzvariablen immer `private` zu deklarieren und Methoden immer `public`:

```

class A extends B {
    private int x;
    public int m() {
        ...
    }
}

...
A y = new A(...);
A.m() // OK
A.x // verboten

```

Konstruktoren Ein Java-Objekt wird mit der Konstruktion `new` erzeugt:

```

Account a = new Account(1000);

```

(Dies entspricht dem Aufruf des Konstruktors `make-account` in Scheme.) Nach Konstruktion eines Objekts wird außerdem eine Initialisierungsmethode aufgerufen, die den gleichen Namen wie die Klasse trägt. Diese Methode hat irreführend ebenfalls den Namen *Konstruktor* und bekommt bei der Deklaration keinen Typ für ihren Rückgabewert. Dieser Methode werden die Parameter des `new`-Konstrukts übergeben:

```

class Account {
    private int balance = -1;

    public Account(int initialBalance) {
        balance = initialBalance;
    }
}

```

```
    }  
    ...  
}
```

Ein Konstruktor kann den Konstruktor der Oberklasse unter dem Namen `super` aufrufen.

`super` Das konzeptuell in ein Objekt eingebettete Objekt der Oberklasse trägt den Namen `super`:

```
class A {  
    public int m() {  
        ...  
    }  
}  
  
class B {  
    public int m() {  
        return super.m() + 5;  
    }  
}
```

`null` `null` ist ein Literal für einen ausgezeichneten Wert, ähnlich der leeren Liste in Scheme. `null` hat einen Referenztyp, und kann an jede Variable mit Referenztyp zugewiesen werden.

Exceptions Java hat ein eingebautes Exception-System, das funktioniert wie das System von Blatt 4, Aufgabe 1. Eine Ausnahmesituation wird signalisiert durch den Befehl:

```
throw e;
```

`e` muß dabei ein Objekt einer Unterklasse von `Exception` ergeben, das mehr Informationen über die Art der Ausnahme erkennt.

Ein Handler für die Exception wird etabliert durch `try`:

```
try {  
    e  
}  
catch (Exception v) {  
    h  
}
```

(Es kann auch mehrere `catch`-Klauseln geben, die jeweils unterschiedliche Unterklassen von `Exception` spezifizieren. Es wird automatisch die passende ausgesucht.)

Dabei speichert `try` seine Continuation; `throw` springt den zuletzt etablierten Handler bei `h` an, der in dieser Continuation abläuft.

```
try {  
    ...  
    if (mike == 0) {  
        throw new Exception("Mike ist ausnahmsweise eine Null.");  
    }  
    ...  
}  
catch (Exception x) {
```

```
    System.out.println(x.getMessage());
}
```

(Es kann auch mehrere `catch`-Klauseln geben, die zwischen verschiedenen Unterklassen von `Exception` unterscheiden.)

Wenn eine Methode (direkt oder indirekt) eine Ausnahmesituation signalisieren kann, so muß dies im Kopf der Methode deklariert werden:

```
public void m() throws IOException {
    ...
    throw new IOException("The hard drive is not where I left it.");
    ...
}
```

Pragmatisch empfiehlt es sich, die Deklaration erst wegzulassen und es dem Java-Compiler zu überlassen, daran zu erinnern.

Packages Java wird mit einer großen Sammlung von eingebauten Klassen geliefert. Diese sind in sogenannte *Packages* unterteilt. So gehört z.B. die eingebaute Klasse `Vector` zum Package `java.util`. Normalerweise muß bei allen Packages außer `java.lang` jeweils der Package-Name mit angegeben werden, also `java.util.Vector`. Wem das zuviel Tipparbeit ist, kann in oben in einer `.java`-Datei schreiben:

```
import java.util.Vector;
```

Dann reicht es in dieser Datei, einfach `Vector` zu schreiben. Alle Klassen eines Packages können mit

```
import java.util.*;
```

zugänglich gemacht werden.

main Es gibt keine REPL. Stattdessen wird der Java-Interpreter auf eine Klasse angewendet, die eine Art Methode `main` mit der Deklaration:

```
public static void main(String[] args)
```

enthalten muß. Diese Methode wird beim Programmstart aufgerufen. (Dafür wird allerdings kein Objekt der Klasse erzeugt, in der die Methode steht.)

2 Sprachkonstrukte im Vergleich

#t, #f	true, false
(and e_1 e_2)	(e_1 && e_2)
(or e_1 e_2)	(e_1 e_2)
(not e)	! e
(eq? e_1 e_2) bzw. (= e_1 e_2)	$e_1 == e_2$
(if t c a)	if (t) { c } else { a } (Befehl, gibt keinen Wert zurück)
(if t c a) (Ausdruck, hat Wert)	$t ? c : a$
(display e)	System.out.print(e)
(display e) (newline)	System.out.println(e)
(make- x e_1 e_2 ... e_n)	new x (e_1, e_2, \dots, e_n)
self	this
eingebettetes Objekt der Oberklasse	super
(ask e 'm p_1 p_2 ... p_n)	$e.m(p_1, p_2, \dots, p_n)$
(ask self 'm p_1 p_2 ... p_n)	$m(p_1, p_2, \dots, p_n)$
(+ e_1 e_2) (ebenso für die anderen arithmetischen Operatoren)	($e_1 + e_2$)
Werrückgabe von e	return e
(begin e_1 e_2 ... e_n)	$e_1; e_2; \dots; e_n$ (hat aber selbst keinen Wert)
(set! x e)	$x = e$
(let* ((v_1 e_1) ... (v_n e_n)) e)	{ t_1 $v_1 = e_1$; ... t_n $v_n = e_n$; e }
	(t_1, \dots, t_n sind die Typen von v_1, \dots, v_n)
(letrec ((loop (lambda () (if w (begin b u (loop)))))) i (loop))	for ($i; w; u$) { b }

```
(letrec
  ((loop (lambda ()
           (if w
               (begin
                  b
                  (loop))))))
  (loop))
```

```
while (w) {
  b
}
```

```
(try (lambda ()
      e)
      (lambda (v)
        h))
```

```
try {
  e
}
catch (Exception v) {
  h
}
```

3 Struktur einer Java-Klasse

Die Struktur einer Java-Klasse lässt sich grob mit der Struktur einer Klasse im Objektsystem aus der Vorlesung vergleichen. Neben leichten Umstrukturierungen ist der Hauptunterschied die Einführung von Typen; dazu später. Eine Java-Klasse hat grob folgende Form:

```
public class c extends s {  
  
    // lokale / Instanzvariablen  
    private  $t_1^i$  v1 = e1;  
    ...  
    private  $t_k^i$  vk = ek;  
  
    // Konstruktor / Initialisierungsmethode  
  
    public c( $t_1^c$  p1c, ...,  $t_m^c$  pmc) {  
        bc  
    }  
  
    // Methoden  
    public  $t_1^r$  m1( $t_{1,1}^p$  p1,1, ...,  $t_{1,n_1}^p$  p1,n_1) {  
        b1  
    }  
    ...  
    public  $t_n^r$  mn( $t_{n,1}^p$  pn,1, ...,  $t_{n,n_n}^p$  pn,n_n) {  
        bn  
    }  
}
```

Beispiel:

```
public class Account {  
    private int balance;  
  
    public Account(int initialBalance)  
    {  
        balance = initialBalance;  
    }  
  
    public int withdraw(int amount)  
    {  
        if (amount <= balance)  
            balance = balance - amount;  
        return balance;  
    }  
  
    public int deposit(int amount)  
    {  
        balance = balance + amount;  
        return balance;  
    }  
}
```

Die dazu passende Scheme-Klasse hat folgende Struktur:

```

(define make-c
  (lambda (p1c ... pmc)
    (let ((super (make-s ...)))
      (let* ((v1 e1)
             ...
             (vk ek))

        bc

        (lambda (message)
          (cond
            ((eq? message 'm1)
             (lambda (self p1,1 ... p1,n1)
               b1))
            ...
            ((eq? message 'mn)
             (lambda (self pn,1 .. pn,nn)
               bn))
            (else (no-method 'c))))))))

```

Beispiel:

```

(define make-account
  (lambda args
    (let ((balance
          (if (null? args)
              100
              (car args))))
      (lambda (message)
        (cond
          ((eq? message 'withdraw)
           (lambda (self amount)
             (if (<= amount balance)
                 (set! balance (- balance amount))
                 balance))
           (lambda (self amount)
             (set! balance (+ balance amount))
             balance))
          (else (no-method 'account)))))))

```

4 Beispiel aus der Vorlesung

```

public class Speaker {
  public void say(String stuff)
  {
    System.out.println(stuff);
  }
}

public class Lecturer extends Speaker {
  public void lecture(String stuff)
  {
    say(stuff);
  }
}

```

```

        say("abstraction abstraction abstraction");
    }
}

public class ArrogantLecturer extends Lecturer {
    public void say(String stuff)
    {
        super.say("it is obvious that " + stuff);
    }
}

```

Zum Vergleich die Scheme-Version:

```

(define make-speaker
  (lambda ()
    (lambda (message)
      (cond ((eq? message 'say)
             (lambda (self stuff)
               (display stuff)
               (newline)))
            (else (no-method 'speaker))))))

```

:: ASK muß entsprechend geändert werden

```

(define ask
  (lambda (object message . args)
    (let ((method (get-method object message)))
      (if (method? method)
          (apply method (cons object args))
          (error "No method" message (cadr method))))))

```

:: Neue Dozenten und arrogante Dozenten

```

(define make-lecturer
  (lambda ()
    (let ((speaker (make-speaker)))
      (lambda (message)
        (cond
         ((eq? message 'lecture)
          (lambda (self stuff)
            (ask self 'say stuff)
            (ask self 'say '(abstraction abstraction abstraction))))
         (else
          (get-method speaker message))))))

```

```

(define make-arrogant-lecturer
  (lambda ()
    (let ((lecturer (make-lecturer)))
      (lambda (message)
        (cond ((eq? message 'say)
               (lambda (self stuff)
                 (ask lecturer 'say
                    (append '(it is obvious that)
                             stuff))))
              (else (get-method lecturer message))))))

```