

Praktische Informatik 1

Variablen, Referenzen, Zuweisungen

Thomas Röfer

- Variablen
- Zuweisung / Initialisierung
- Konstanten
- Lebenszeit / Sichtbarkeit
- Java Stack / Heap
- Call by Reference / Value
- Musterlösung, Übungsblatt

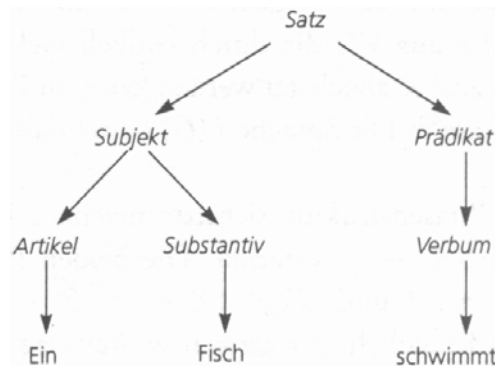


Rückblick „Grammatik, Java-Programme, Bezeichner, Datentypen“

Syntax/Semantik



Chomsky-Grammatiken



EBNF

Produktion = Bezeichner '=' Ausdruck
 Ausdruck = Term { '|' Term }
 Term = Faktor { Faktor }
 Faktor = Bezeichner |
 '\" Terminal '\" |
 '(' Ausdruck ')' |
 '[' Ausdruck ']' |
 '{' Ausdruck '}'

Eigenständige Programme

>java Klasse Hallo 21333 "dies und das"

String[] args	
args.length	3
args[0]	"Hallo"
args[1]	"21333"
args[2]	"dies und das"

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

Basisdatentypen

byte	1 Byte	-128 bis 127
short	2 Bytes	-32768 bis 32767
int	4 Bytes	-2147483648 bis 2147483647
long	8 Bytes	-9223372036854775808 bis 9223372036854775807
float	4 Bytes	±1.40239846E-45 bis ±3.40282347E+38
double	8 Bytes	±4.94065645841246544E-324 bis ±1.79769313486231570E+308
boolean	1 Byte	false, true
char	2 Bytes	'\u0000' bis '\uFFFF'

Hüllklassen

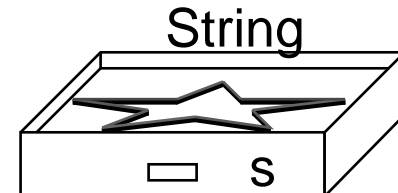
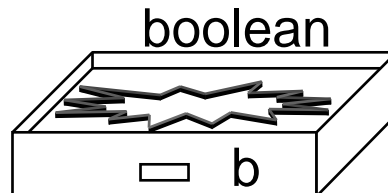
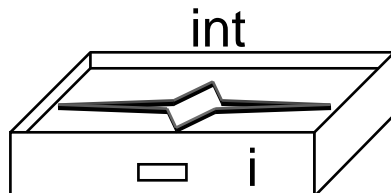
Datentyp	Hüllklasse
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Char

Variablen, Objekte und Referenzen

- Speicherstellen haben
 - eine Adresse
 - einen Inhalt
- Variablen haben
 - einen Namen (i, b, s, ...)
 - einen Typ (int, boolean, String, ...)
 - einen Wert (7, true, "Hallo", ...), d.h. den Inhalt der Speicherstelle
 - einen Ort, an dem ihr Wert gespeichert wird, d.h. die Referenz auf die Speicherstelle
 - eine Lebenszeit

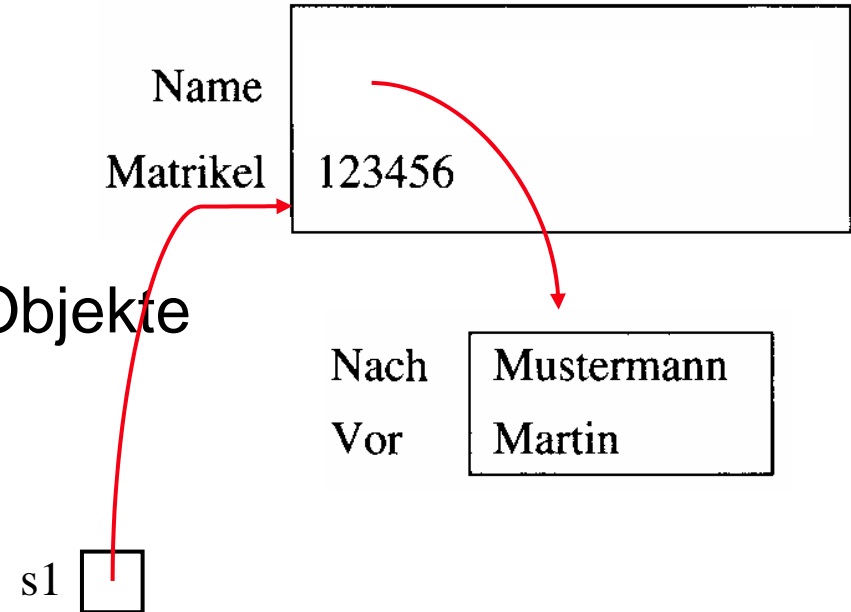
Adresse Inhalt

0	15
1	254
2	0
3	7
5	42



Variablen, Objekte und Referenzen

- Objekte haben in Java
 - keinen Namen
 - sonst alles, was auch Variablen haben
 - Reihungen sind spezielle Objekte
- Referenzen
 - zeigen auf Speicherstellen
- Referenzen in Java
 - sind Variablen
 - Ihr Wert zeigt auf den Ort, an dem ein Objekt gespeichert ist
 - oder auf *null*



Deklaration von Variablen

- Lokale Variablen in Funktionen
 - Existieren 1x pro Aufruf der Funktion
 - *void fun() { int n; ... }*
- Parameter von Funktionen
 - Sind auch lokale Variablen
 - Existieren 1x pro Aufruf der Funktion
 - *int factorial(int n) { ... }*
- Objektattribute
 - Existieren 1x pro Objekt (instancierter Klasse)
 - *class Name { String vorname; ... }*
- Klassenattribute
 - Existieren 1x pro Klasse (unabhängig von Instanzen)
 - *class System { static PrintStream out; ... }*

Zuweisungen

- Eine Zuweisung
 - ändert den Wert einer Variablen
 - ist ein Ausdruck
Linkswert = Rechtswert
- Linkswert (L-Wert, L-Value)
 - Ein Wert, der in einer Zuweisung links vom = stehen kann
 - Muss sich immer zu einer Referenz auf eine Speicherstelle auswerten

```
Vorlesung v = new Vorlesung();  
v.finde("Martin Mustermann").punkte[9] = 95;
```

```
int a;  
a = 17;  
17 = a; // Fehler
```

```
class Student {  
    int[] punkte = new int[12];  
}  
  
class Vorlesung {  
    Student finde(String name) {  
        :  
    }  
}
```

Zuweisungen

- Rechtswert (R-Wert, R-Value)
 - Ein Wert, der in einer Zuweisung rechts stehen kann
 - R-Werte können auch als Funktionsparameter benutzt werden
 - L-Werte sind immer auch R-Werte
 - Umgekehrt gilt das nicht!

```
int a;  
a = 17;  
a = 17 * 29 * a + factorial(5 + 7);
```

Initialisierung

- Eine Initialisierung setzt den anfänglichen Wert einer Variablen
- Parameter von Funktionen
 - Initialisierung durch übergebenen Wert

```
void f(int n) { ... }  
f(15);
```

- Parameter n bekommt den Wert 15
 - Äquivalent zu $n = 15$;
- Lokale Variablen
 - Werden nicht automatisch initialisiert
 - Aber Compiler erzwingt Initialisierung vor Benutzung als R-Wert

```
void f() {  
    int m;  
    int n = m; // Fehler  
}
```

```
String toString(boolean b) {  
    String s;  
    if (b == true) {  
        s = "true";  
    } else if (b == false) {  
        s = "false";  
    }  
    return s; // Fehler  
}
```

Initialisierung

- Klassen und Objektattribute
 - Automatische Initialisierung
 - Zahlen mit 0
 - Wahrheitswerte mit *false*
 - Referenzen mit *null*
 - Manuelle Initialisierung
 - Im Konstruktor (bei Objektattributen)
 - Direkt bei der Deklaration
- Elemente von Reihungen
 - Automatische Initialisierung wie bei Objekten
 - Manuelle Initialisierung durch Aufzählung der Elemente

```
class Student {  
    Name name;  
    int matrikel;  
    int[] punkte = new int[12];  
  
    Student(Name n, int m) {  
        name = n;  
        matrikel = m;  
    }  
}
```

```
char[][] a = {  
    {'A', 'B', 'C'},  
    {'D', 'E'},  
    {'F', 'G', 'H', 'I'},  
};
```

Konstanten (finals)

- Konstanten ändern ihren Wert nach ihrer Initialisierung niemals
- Übersetzungszeit-Konstanten
 - lassen sich zur Übersetzungszeit berechnen
 - sind benannte Literale, können also überall verwendet werden, wo auch Literale zulässig sind
 - Hinter *case* in der *switch*-Anweisung
 - Beim Initialisieren von Reihungsattributen
 - werden gleich bei ihrer Deklaration initialisiert

```
boolean isOdd(int n) {  
    final int EVEN = 0;  
    final int ODD = 1;  
    switch (n % 2) {  
        case EVEN:  
            return false;  
        case ODD:  
            return true;  
    }  
}
```

```
class Student {  
    final int AUFGABEN = 12;  
    int[] punkte = new int[AUFGABEN];  
}
```


Konstanten (finals)

- Laufzeit-Konstanten
 - lassen sich erst zur Laufzeit errechnen
 - sind „schreibgeschützte“ Variablen, d.h. keine Literale
 - Können nach ihrer Deklaration initialisiert werden
 - Können in jeder neuen Lebenszeit einen anderen Wert bekommen
- Konstante Referenzen können auf veränderliche Objekte zeigen

```
class Constant {  
    final int MAX_STUDENTS;  
    Constant(int students) {  
        MAX_STUDENTS = students;  
    }  
}
```

```
void f(int[] a) {  
    int i = 0;  
    while (i < a.length) {  
        final int val = a[i];  
        :  
    }  
}
```

```
void f() {  
    final int[] a = new int[1];  
    a[0] = 1;  
    a = new int[2]; // Fehler!  
}
```

Lebenszeit von Variablen

- Lokale Variablen
 - Bis Ende des aktuellen Blocks {...}
 - Ausnahme: Definitionen in for
 - for (int i = 0; i < 10; ++i) {...}ist eigentlich
{ int i; for (i = 0; i < 10; ++i) {...} }
- Funktionsparameter
 - Bis zum Ende des Funktionsrumpfes
 - void f(int a) {...} ist eigentlich
void f { (int a) {...} }

```
int f(int a) {  
    int b;  
    if (a == 0) {  
        int c = 1;  
        b = c;  
        // Ende von c  
    } else {  
        int c = a * f(a - 1);  
        b = c;  
        // Ende von c  
    }  
    return b;  
    // Ende von a und b  
}
```


Lebenszeit von Klassenattributen

```
class A {  
    static A a = new A();  
  
    A() {  
        System.out.println("A()");  
    }  
  
    static void test() {  
        System.out.println("A.test()");  
    }  
}
```

```
class B {  
    static void test() {  
        System.out.println("B.test()");  
        A.test();  
    }  
}
```

- Von der ersten Benutzung der Klasse bis zum Programmende

Lebenszeit von Objekten und Reihungen

- Beginnt mit dem Erzeugen durch *new*
- Endet, wenn der *Garbage Collector* sie freigibt
 - wenn keine Referenz mehr auf sie zeigt
- Der *Garbage Collector* läuft an, wenn
 - Speicher knapp wird
 - das Programm nichts anderes zu tun hat
 - *System.gc()* aufgerufen wird
- Vor dem Entfernen eines Objekts wird die Objektmethode *finalize()* aufgerufen
 - Wird selten benötigt

```
class A {  
    protected void finalize() {  
        System.out.println(  
            "Letzter Atemzug...");  
    }  
}
```

Lebenszeit von Objekten und Reihungen

```
class C {  
    C() {  
        System.out.println("C()");  
    }  
  
    protected void finalize() {  
        System.out.println("C.finalize()");  
    }  
}
```

```
class D {  
    static void test(int n) {  
        System.out.println("D.test()");  
        C c = new C();  
        System.out.println("c = null");  
        c = null;  
        while (n > 0) {  
            int[] a = new int[1000];  
            n = n - 1;  
        }  
        System.out.println("System.gc()");  
        System.gc();  
    }  
}
```

Sichtbarkeit

- Sichtbarkeit, suchen nach Bezeichner
 - In Funktion rückwärts, aber nicht in Blöcke hinein
 - In Funktionsparametern
 - In Klasse (auch hinter der aktuellen Funktion)
- Mehrfachdefinitionen in Funktionen sind verboten

```
class HideAndSeek {  
    static int target;  
    static void fun() {  
        if (target > 10) {  
            System.out.println(target);  
            int target = 17;  
            System.out.println(target);  
        }  
        System.out.println(target);  
    }  
}
```

```
void f(int a) {  
    int a  
    {  
        int a;  
    }  
}
```



Sichtbarkeit

- Beim Zugriff auf
 - Klassenattribute und Methoden *dieser Klasse* verhält sich Java so, als wenn „*Klassenname*.“ vor dem Bezeichner stehen würde
 - Objektattribute und Methoden *dieses Objekts* verhält sich Java so, als wenn „*this*.“ vor dem Bezeichner stehen würde
- Zugriff auf gerade nicht sichtbare Variablen
 - Objektattribute: *this.variable*
 - Klassenattribute: *Klassenname.variable*

```
class E {  
    static int a;  
    int b;  
    void test1() {  
        a = b;  
    }  
    void test2() {  
        E.a = this.b;  
    }  
    void test3(int a, int b) {  
        E.a = a;  
        this.b = b;  
    }  
}
```

Reihungen (arrays)

- Sind Objekte
 - Man kann aber keine eigenen Reihungsklassen implementieren
- Sie enthalten ein konstantes *int*-Attribut *length*, das ihre Länge angibt
- Ein Zugriff außerhalb der Array-Grenzen $[0 \dots \text{length}-1]$ erzeugt eine *ArrayIndexOutOfBoundsException*
- Mehrdimensionale Reihungen sind Reihungen von (Referenzen auf) Reihungen
 - Sie können weiteren Dimensionen unterschiedliche Größen haben

Index	0	1	2	3
0	A	B	C	
1	D	E		
2	F	G	H	I


```
char[][] a = {  
    {'A', 'B', 'C'},  
    {'D', 'E'},  
    {'F', 'G', 'H', 'I'},  
};
```


Stapel und Halde in Java

- Stapel (Stack)
 - Speichert Funktionsparameter, lokale Variablen und Rücksprungadressen
 - Wächst und schrumpft mit Funktionsaufrufen
 - Daten liegen immer zusammenhängend vor
- Halde (Heap)
 - Speichert Objekte und Reihungen
 - Speicher auf dem Heap kann in beliebiger Reihenfolge belegt und freigegeben werden
 - Daten können verstreut auf dem Heap liegen
 - Heap wird durch *Garbage Collection* aufgeräumt

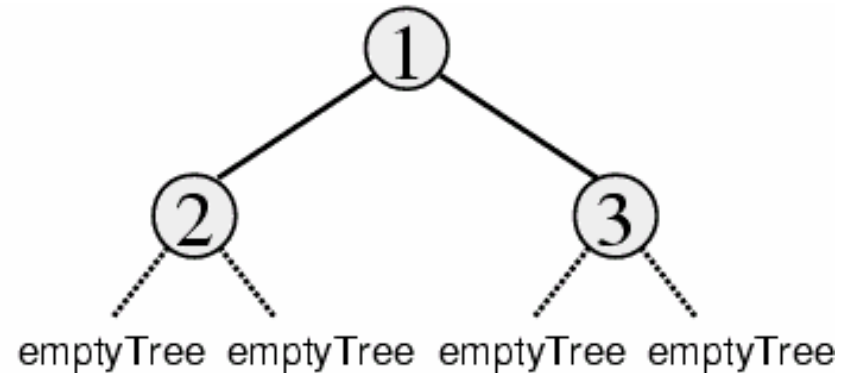
Java-Stack (Stapel, Keller)

```
class Foo {  
    static int main(int j) {  
        int i = 1;  
        int k = fun1(i, j);  
        return k;  
    }  
  
    static int fun1(int a, int b) {  
        boolean x;  
        return a * b;  
    }  
}
```



Beispiel: BinTree

```
class BinTree {  
    BinTree left;  
    int val;  
    BinTree right;  
    boolean isEmpty;  
  
    BinTree() {isEmpty = true;}  
  
    BinTree(BinTree l, int v, BinTree r) {  
        left = l;  
        val = v;  
        right = r;  
        isEmpty = false;  
    }  
}
```



```
BinTree b = new BinTree(  
    new BinTree(new BinTree(), 2, new BinTree()),  
    1,  
    new BinTree(new BinTree(), 3, new BinTree())  
);
```