Assignment is undoubtedly the most characteristic feature of programming a digital computer, and one that most clearly distinguishes it from other branches of mathematics.

C. A. R. Hoare (1969)

Variablen

Referenzen

Zuweisungen

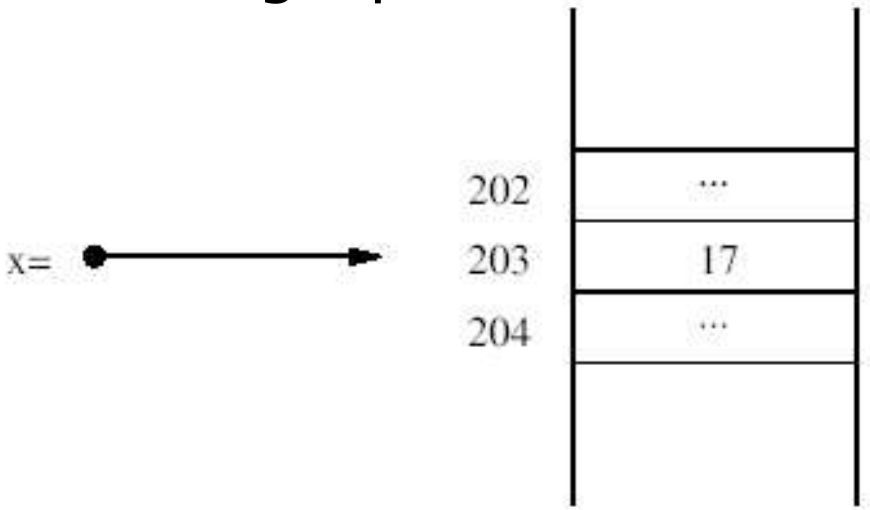
Variablen

- Variablen dienen der Speicherung von Werten
  - **Name** (*name*) bezeichnet die Variable im Programm
  - **Wert** (*value*) ist das Datenelement, dass in der Variablen gespeichert ist
  - **Typ** (*type*) legt den Typ möglicher Werte fest
    - Basistypen
    - Objekttypen (Klassen)
    - Arraytypen

Referenztypen

- Objekte häufig mit der Speicherstelle identifiziert, an der sie gespeichert sind
- Variablen, die eine Speicherstelle beinhalten heißen Referenzen / Zeiger
- Darum oft:
  Objekttyp = Referenztyp

- Beispiel:
  Variable x hat als Wert Objekt, das bei Stelle 203 gespeichert ist.



Referenztypen

- **Achtung**
  Die Identifikation von Objekt und Referenz ist
  - anschaulich
  - aber eigentlich falsch,
    denn verschiedene Abstraktionsebenen werden vermengt, was wieder zu Verwirrung führen kann
    - Objekt: abstraktes Gebilde
    - Referenz (auf Speicherstelle): konkrete Implementierung des abstrakten Gebildes

Java Arithmetik

Elementare Ganzzahltypen

- Zahlbereiche der Ganzzahltypen

Typ	Bits	Minimalwert	Maximalwert
`byte`	8	-128	127
`short`	16	-32.768	32.767
`int`	32	-2.147.483.648	2.147.483.647
`long`	64	-9.223.372.036.854.775.808	9.223.372.036.854.775.807
`char`	16	0 (`'\u0000'`)	65.535 (`'\uffff'`)

Konversionsmethoden in Hüllklassen

- Konversionsmethoden in Hüllklassen

Typ	Konversion aus String
byte	Byte.parseByte(String)
short	Short.parseShort(String)
int	Integer.parseInt(String)
long	Long.parseLong(String)
float	Float.parseFloat(String)
double	Double.parseDouble(String)

Konversionsmethoden in Hüllklassen

- **Beispiel**:

Rahmenprogramm Program2 gestartet mit der Kommandozeile

```
java Program2 123 456
```

Dann innerhalb von main

- in der Variablen args[0] die Zeichenkette "123"
- in der Variablen args[1] die Zeichenkette "456"

Entsprechende Ganzzahlen in den Variablen int x, y; erhält man durch

```
x = Integer.parseInt(args[0]);
y = Integer.parseInt(args[1]);
```

Ganzzahlarithmetik

- **Java Ganzzahl-Arithmetik** ist
  - **Zweierkomplement-Arithmetik**
  - **modulo** dem Darstellungsbereich
- Bei "Überlauf" keine Fehlermeldung (Exception)
- Achtung:
  Unerwünschter Überlauf bleibt unbemerkt!
- Der Darstellungsbereich ist zu einem „Ring" geschlossen
- Beispiel:
  größte positive Zahl + 1 == kleinste negative Zahl

- **Ganzzahldivision**
  - Operator in Java:
    - / für Ganzzahldivision
    - % für Rest bei Ganzzahldivision
  - Achtung:
    - 1/2 + 0.5 == 0.5 , aber
    - 1.0/2 + 0.5 == 1.0
  - Java rundet immer zur Null hin, um Ganzzahlen zu erhalten:
    - 3/2 == 1
    - -3/2 == -1

- **Ganzzahldivision**
  - Für Ganzzahlen gilt

    (x/y) * y + x%y == x

  - Beispiele:
    - 3 %2 == 1
    - (-3) %2 == -1

Gleitkommaarithmetik

- Java Gleitkomma-Arithmetik implementiert den Standard **IEEE 754-1985**
  - **Einfach genaue Zahlen: float**
  - **Doppelt genaue Zahlen: double**
- Arithmetik kann
  - nach $+\infty$ und $-\infty$ überlaufen (**overflow**)
    Zahlen werden absolut zu groß
  - nach +0.0 oder -0.0 unterlaufen (**underflow**)
    Zahlen werden absolut zu klein
- Es gibt eine positive und eine negative Null, wobei jedoch +0.0 == -0.0

Gleitkommaarithmetik

- Der Wert **NaN** (not a number) repräsentiert Rechnungen, die keinen sinnvollen Wert haben
  - Beispiele:
    - 0 * (+∞) == NaN
    - (+∞) + (-∞) == NaN
    - 0 / 0 == NaN
    - Aber:
      (+∞) + 5 == (+∞)
      1 / (+0.0) == (+∞)
  - Ausdrücke, in denen NaN vorkommt, liefern immer NaN als Ergebnis

Gleitkommaarithmetik

- Sonderfälle für Gleitkommadivision

x	y	x/y	x % y
Finite	±0.0	±∞	NaN
Finite	±∞	±0.0	x
±0.0	±0.0	NaN	NaN
±∞	Finite	±∞	NaN
±∞	±∞	NaN	NaN

Operatoren und Ausdrücke

Zuweisungsoperatoren

- In Java haben **Zuweisungen** einen **Wert**
  - Wert einer Zuweisung ist der Wert ihrer rechten Seite
  - Zuweisungen können als (Teil-)Ausdrücke verwendet werden
  - Änderung der Variable ist dann **Seiteneffekt** der Auswertung
- Methodisch problematisch, da
  - Ausdrücke (expressions) und Anweisungen (statements) vermischt
  - kurzer aber unübersichtlicher Code

Zuweisungsoperatoren

- x = y = 1
  - was das gleiche ist wie x = (y = 1)
  - Wert von y = 1 ist 1, was an x zugewiesen wird
  - Als Seiteneffekt der Auswertung wird 1 an y zugewiesen
- x = (y = 1) + 5
  - hat den Wert 6
  - weist als Seiteneffekt
    - y den Wert 1
    - x den Wert 6

Zuweisungsoperatoren

- Zuweisungsausdrücke der Form

  v = v+expr

  können in Java auch in der Form

  v += expr

  geschrieben werden
- Für jeden binären Operator gibt es einen entsprechenden kombinierten Zuweisungsoperator
  - Etwa -= *= /= %= usw.
- Effekt von v = v ° expr und v °= expr (zumeist!) gleich

Zuweisungsoperatoren

- **Achtung**:
  - Bei v °= expr wird v nur einmal ausgewertet
  - Bei v = v ° expr zweimal
- Falls v selbst ein Ausdruck ist, dessen Auswertung Seiteneffekte hat, kann dies zu einem Unterschied führen
- Beispiel für Unterschied:
  - a[i++] = a[i++] + 1
  - a[i++] += 1
- Es ist guter Programmierstil, solche Fälle zu vermeiden

Arithmetische Operatoren

- Für die elementaren Zahltypen:
  - unäre Operationen + und -,
  - binären Operationen +, -, *, / und %
- Auf den Ganzzahltypen ist / die Ganzzahldivision, auf Fließkommatypen die „übliche“ Division
- Die Divisionsrest-Operation % existiert auch auf Fließkommatypen
  - Beispiel: 7.0 % 2.5 == 2.0

Inkrement und Dekrement-Operatoren

- Zusätzlich unäre Operatoren ++ und --
  - Können in Postfix und Präfixform verwendet werden
  - **Präfixform:**
    Seiteneffekt geschieht **vor** Auswertung der Variablen
  - **Postfixform:**
    Seiteneffekt geschieht **nach** Auswertung der Variablen

Inkrement und Dekrement-Operatoren

- Ausdruck i-- (bzw. i++)
  - Wert ist der Wert der Variablen i
  - Als **Seiteneffekt** der Auswertung wird i um Eins erniedrigt (bzw. erhöht)
- Ausdruck --i (bzw. ++i)
  - Wert ist i-1 (bzw. i+1)
  - Als **Seiteneffekt** wird der Wert von i um Eins erniedrigt (bzw. erhöht)

Inkrement und Dekrement-Operatoren

- Es hat sich eingebürgert (guter Stil!), im Zweifel die Postfixform i++ zu benutzen
- Man schreibt immer

```
i++;
```

  statt der gleichwertigen Anweisungen
  - i += 1;
  - i = i+1;
- Solche „Standardformen“ heißen auch **Idiom**

Boolesche Operatoren

- In Java wird der Typ boolean mit den Booleschen Literalen true und false zur Verfügung gestellt
- Als **logische Operatoren** existieren

&	logisches UND ($\wedge$)
|	logisches ODER ($\vee$)
^	logisches EXKLUSIVES ODER (XOR)
!	logische Negation ($\neg$)
&&	bedingtes logisches UND ($\wedge$)
||	bedingtes logisches ODER ($\vee$)

Boolesche Operatoren

- && und || heißen **bedingte** logische Operatoren, da sie ihren rechten Operanden nur dann auswerten, falls dies wirklich nötig ist
- Beispiel: In

  ((b = false) && (c = true))

  wird die Zuweisung c = true nicht ausgeführt, da die linke Seite b = false zu false auswertet und damit der Wert des Gesamtausdrucks schon als false feststeht
- Die symmetrischen binären Operatoren &, | und ^ evaluieren jeweils beide Operanden

Boolesche Werte und Vergleichsoperatoren

- Boolesche Werte werden oft durch Vergleiche erzeugt
- In Java gibt es die üblichen **Vergleichsoperatoren**

>	größer
>=	größer oder gleich
<	kleiner
<=	kleiner oder gleich
==	gleich
!=	ungleich

Beispiele:
(x%2 == 0) liefert true, falls ein ganzzahliges x geradzahlig ist, und false sonst.

((a >= b) || (a <= b)) && (a !=a) evaluiert zu false

Vergleich Boolescher Werte

- **Bemerkung**:
  - Es ist schlechter Stil, Vergleichsoperatoren auf Boolesche Werte anzuwenden
  - Daraus könnten schwer zu findende Fehler entstehen (a = true statt a == true)
  - Man schreibt also

    (a && !b)

    statt

    (a == true && b == false)

Syntax von Ausdrücken

- Ausdrücke können (zunächst) mehrdeutig sein
  - 1+2*3 könnte man syntaktisch lesen als
    - (1+2)*3 oder
    - 1+(2*3)
- Wir brauchen aber eindeutige Semantik
- Daher hat jeder Operator eine **Bindungskraft** (auch: Vorrang, **Präzedenz**, *precedence*)
- Beispiel: * bindet stärker als +
  Daher: 1+2*3 == 1+(2*3)

Syntax von Ausdrücken

- **Präzedenzen** von Java-Operatoren (absteigend)

Postfix Operatoren	`[] . (params) expr++ expr--`
Unäre Operatoren	`++expr --expr +expr -expr ~ !`
Erzeugung und Anpassung	`new (type) expr`
Multiplikative Op.	`* / %`
Additive Op.	`+ -`
Schiebe-Op.	`<< >> >>>`
Relationale Op.	`< > >= <= instanceof`
Gleichheits-Op.	`== !=`
Bitweises und log. UND	`&`
Bitweises und log. XOR	`^`
Bitweises und log. ODER	`|`
Bedingtes logisches UND	`&&`
Bedingtes logisches ODER	`||`
Bedingung	`? :`
Zuweisungs-Operator	`= += -= *= /= %= >>= <<= >>>= &= ^= |=`

Syntax von Ausdrücken

- Mehrere Vorkommnisse von Operatoren gleicher Präzedenz (z.B. 1+2+3 oder x=y=1) werden durch Angabe der **Assoziativität** (*associativity*) eindeutig:
- Alle **binären Operatoren** sind **links-assoziativ**, außer den **Zuweisungsoperatoren**, die **rechts-assoziativ** sind
- Beispiele:
  - 1+2+3 steht für (1+2)+3
  - x=y=1 steht für x=(y=1)

Syntax von Ausdrücken

- Operator-Präzedenzen so gewählt, dass sich keine Überraschungen ergeben
- Aber Achtung: Gerade bei wenig gebrauchten Operatoren doch Überraschungen. Solche Fehler sind schwer zu finden.
- Im Zweifel immer Klammern verwenden!
- Beispiel:

  (a & b == b & c) hat die Bedeutung von

  (a & (b==b)& c)

  da == stärker bindet als symmetrisches UND &

Typkonversionen

- Bisher sind wir davon ausgegangen, dass alle Ausdrücke ganz genau typkorrekt aufgebaut sind
- Wenn ein Operator ein Argument a vom Typ T verlangt hat, dann auch nur Ausdrücke vom Typ T für a eingesetzt.
- Grundprinzip **stark typisierter** Sprachen (*strongly typed languages*) wie Java
- Allerdings in dieser Strenge hinderlich
  Beispiele:
  - Sei int x; long y; dann y = x nicht typkorrekt
  - 2+2L nicht typkorrekt

Typkonversionen

- **Automatische Typanpassung**, wo dies problemlos geschehen kann
- Genauer:
  wenn ein Ober- und ein Untertyp aufeinander treffen,
  kann Wert des Untertyps als Wert des Obertyps aufgefasst werden

Typkonversionen

- Bei numerischen Typen:
  - Typen mit größerem Wertebereich sind jeweils Obertyp
    - byte $\subseteq$ short $\subseteq$ int $\subseteq$ long
    - char als Ganzzahl
    - float $\subseteq$ double
    - long $\subseteq$ float
  - Bei arithmetischen Operatoren werden die Argumente immer mindestens(!) auf int erweitert

Typkonversionen

- Beispiele: Der Typ von
  - 1 + 1.0f ist float
  - 1 + 1.0 ist double
  - 1L + 1.0f ist float
  - 1 + 1L ist long
  - Sei byte a,b,c; gegeben

    a + b ist int

    –b ist int

    Daher sind folgende Zuweisungen fehlerhaft
    - a = -b;
    - c = a + b;

Explizite Typkonversionen

- Neben den automatischen (impliziten) Typkonversionen sind auch **explizite Typkonversionen** möglich
- **Konversionsoperator** (Type Cast)
  (type) expr,
  wandelt den Typ des Ausdrucks expr zu type
- Explizite Konversionen in der Regel vom Ober- zum Untertyp (Typverengung)
- Nicht jede Typkonversion ist möglich
  Bsp.: Konversion von boolean nach int nicht möglich

Explizite Typkonversionen

- Bei Ganzzahltypen werden die wegfallenden höherwertigen Bits einfach ausmaskiert
  - Signifikante Bits fallen weg
  - Hierbei kann sich ein positiver zu einem negativen Wert wandeln
- Von double zu float kann
  - Präzision verloren gehen
  - eine *infinity* entstehen
- Bei der Konversion von Gleitkommatypen zu Ganzzahltypen entfallen die Nachkommastellen durch Rundung zur Null hin

Explizite Typkonversionen

- **Beispiele:**
  - int x = (int) 1L;
    - Der Wert 1 vom Typ long wird zum Wert 1 vom Typ int umgewandelt
  - char z = (char) 127;
    - Der Wert 127 vom Typ int wird zu einem Zeichenwert umgewandelt (der das DEL-Zeichen repräsentiert)
  - byte c = (byte) (a + b);
    - Der Ausdruck ist korrekt

Explizite Typkonversionen

- **Beispiele (Forts.)**

`(byte) 128` ist -128.	`(byte) 256` ist 0.
`(byte) 129` ist -127.	`(byte) 257` ist 1.
`(byte) 127` ist 127.	`(byte) 255` ist -1.
`(byte) -128` ist -128.	`(byte) -13.5` ist -13.

Typkonversion bei geschachtelten Ausdrücken

- Sind bei **geschachtelten** Ausdrücken Typanpassungen vorzunehmen, so geschieht dies **von innen nach außen**
- Führt manchmal zu Überraschungen!
- Beispiele:
  - double d = 1/2; initialisiert d mit 0.0
    double d = 1.0/2; initialisiert d mit 0.5
  - 0.8 + 1/2 evaluiert zu 0.8
    0.8 + 1.0/2 evaluiert zu 1.3