Ludwig-Maximilians-Universität München
Institut für Informatik
Prof. Dr. Peer Kröger
Michael Fromm, Florian Richter

Einführung in die Programmierung

WS 2018/19

Übungsblatt 7: Typecast, Schleifen

Aufgabe 7-1 *Typecasting*

(a) Geben Sie für jeden der folgenden Java-Ausdrücke an, ob er syntaktisch korrekt ist. Falls ja, geben Sie außerdem den Wert und den Typ des Ausdrucks an. Begründen Sie jeweils Ihre Antwort. Hierbei seinen die Variablen `x, y` und `z` wie folgt initialisiert: `int x = 9, y = 12, z = 16;`

(i) `10/4.`
(ii) `10/4`
(iii) `x == y ? x < y : y > z`
(iv) `false && true`
(v) `1 || 0`
(vi) `(byte)(127+1)`
(vii) `'x' + y + z`
(viii) `x + y + "z"`

Lösungsvorschlag:

(i) syntaktisch korrekt; 2.5 vom Typ double. (impliziter Typecast zu double)
(ii) syntaktisch korrekt; 2 vom Typ int. (Integerdivision)
(iii) syntaktisch korrekt; false vom Typ boolean.
(`((9 == 12) = false => 2.Fall (y>z): (12 > 16)=false`)
(iv) syntaktisch korrekt; false vom Typ boolean.
(„schnelle Verundung“ = wenn der erste Teil schon false ist, dann wird der 2. Teil nicht ausgewertet.)
(v) syntaktisch inkorrekt, da nur Wahrheitswerte mit dem ODER verglichen werden dürfen. Ansonsten („schnelle Veroderung“).
(vi) syntaktisch korrekt; -128 vom Type byte.
($127_{10} + 1_{10} = 01111111_2 + 00000001_2 = 10000000_2 = -128_{10}$, die größte neg. Zahl)
(vii) syntaktisch korrekt; 148 vom Typ int.
(`'x' + y + z` $= 120 + 12 + 16 = 148$, Typ: int; impliziter Typecast von 'x' zu 120, vgl. ASCII-Zeichentabelle)
(viii) syntaktisch korrekt; 21z vom Typ String.
(`x + y + "z"` $= (9 + 12) \circ$ "z" $= 21z$)

(b) Ausdrücke in Java, die aus mehr als nur einem elementaren Datentyp zusammengesetzt sind, werden je nach Verwendung durch implizite oder explizite Typkonversion ausgewertet.
Ersetzen Sie für jeden der folgenden Java-Ausdrücke schrittweise alle impliziten Typecasts durch Explizite. Geben Sie dann den Wert und den Typ des Ausdrucks an. Falls der Ausdruck dann nicht typkorrekt ist, begründen Sie den Fehler.
Hierbei seien die Variablen `x` und `y` wie folgt initialisiert: `int x = 9, y = 12;`

Bsp: `5L * 3.0` $\Leftrightarrow \Big((double)(5L) * 3.0\Big) \Leftrightarrow \Big(5.0 * 3.0\Big) \Leftrightarrow \Big(15.0\Big)$, vom Typ double.

(i) `(x > y) < true`

(ii) `(long)(3F/4.0) == (int)(4/5.0D)`

(iii) `(long)3F/4.0 == (int)4/5.0D`

(iv) `(int) ((char) 65536)`

(v) `1L + (float) 3D`

(vi) `'4' + '7'`

Lösungsvorschlag:

(i) `(x > y) < true`
nicht typkorrekt. Boolean-Werte können nicht mit dem Kleiner-/Größer-Komparator verglichen werden.

(ii) `(long)(3F/4.0) == (int)(4/5.0D)`
$\Leftrightarrow \Big((long)((double)(3F)/4.0) == (int)((double)4/5.0D)\Big) \Leftrightarrow \Big((long)0.75 == (int)0.8\Big) \Leftrightarrow \Big(0L == 0\Big) \Leftrightarrow \Big(0L == (long)0\Big) \Leftrightarrow \Big(0L == 0L\Big) \Leftrightarrow true$, Typ: boolean.

(iii) `(long)3F/4.0 == (int)4/5.0D`
$\Leftrightarrow \Big(3L/4.0 == 4/5.0)\Big) \Leftrightarrow \Big((double)(3L)/4.0 == (double)4/5.0\Big) \Leftrightarrow \Big(3.0/4.0 == 4.0/5.0\Big) \Leftrightarrow \Big(0.75 == 0.8\Big) \Leftrightarrow false$, Typ: boolean.

(iv) `(int) ((char) 65536)` $\Leftrightarrow \Big((int)\text{'NUL'}\Big) \Leftrightarrow 0$, Typ: int.
Jedes char wird aus 2 Byte zusammengesetzt. D.h. der größte darzustellende char ist $2^{16} - 1 = 65535$, und daher ist $65536 == 0$.

(v) `1L + (float) 3D`
$\Leftrightarrow \Big(1L + 3F\Big) \Leftrightarrow \Big((float)(1L) + 3F\Big) \Leftrightarrow \Big(1F + 3F\Big) \Leftrightarrow 4F$, Typ float.

(vi) `'4' + '7'`
$\Leftrightarrow \Big((int)('4') + (int)('7')\Big) \overset{ASCII-Tabelle}{\Leftrightarrow} \Big(52 + 55\Big) \Leftrightarrow 107$, Typ int.

(c) Gegeben sind nun folgende Variablendeklarationen:

```
long l1 = 126L;        byte b1 = 100;          char c1 = '!';
int i1 = 6;            float f1 = 2.0f;        char c2 = 'c';
short s1 = 8;          double d1 = 0.125;      String str1 = "789";
```

Ersetzen Sie alle impliziten Typecasts durch Explizite. Geben Sie dann an welchen Typ und welchen Wert die folgenden Ausdrücke haben.

(i) (f1 * 3.0f) / (b1 * i1)

(ii) "1 + 2 + 3 = "+ (i1 - 2)

(iii) "l1 + i1 = "+ 126 + 6

(iv) f1 / d1 + l1

(v) c2 / c1

(vi) i1 + str1 + s1

Lösungsvorschlag:

(i) (f1 * 3.0f) / (b1 * i1)
$= 6.0f/(float)((int)(b1) * i1) = 6.0f/600.0f =$ $0.01f$, float als allgemeinster Typ.

(ii) "1 + 2 + 3 = "+ (i1 - 2)
$\Leftrightarrow$ "1 + 2 + 3 = " + $(String)(4) \Leftrightarrow$ "1 + 2 + 3 = " + "4" $\Leftrightarrow$ "1 + 2 + 3 = 4", Typ String.

(iii) "l1 + i1 = "+ 126 + 6
$\Leftrightarrow$ "$l1 + i1 =$ " + $(String)(126) + (String)(6) \Leftrightarrow$ "$l1 + i1 = 1266$", Typ String.

(iv) f1 / d1 + l1
$= (double)(2.0f)/0.125 + (double)(l1) = 16.0 + 126.0 =$ 142.0, Typ: double (d1) als allgemeinster Typ

(v) c2 / c1
Automatische Konvertierung von char in int in arithmetischen Ausdrücken. int-Wert von 'c' ist 99, int-Wert von '!' ist 33 char basierend auf dem Unicode-Zeichensatz.
$= (int)(c2)/(int)(c1) = 99/33 =$ 3, Typ int.

(vi) i1 + str1 + s1
$= (string)(i1) + str1 + (string)(s1) = (string)(6) + str1 + (string)(8) =$ 67898, Typ String.

Hinweise:

- *Zeichen* vom Typ `char` werden in zusammengesetzten arithmetischen Ausdrücken automatisch in den Typ `int` konvertiert (ensprechend ihrem ASCII-Code).
- In Java werden *Zeichenketten* mit doppelten Anführungszeichen umrahmt und sind vom Typ `String`. Die Konkatenation von Zeichenketten erfolgt durch den `+`-Operator (der Operator ist also *überladen*). Ist wenigstens einer der beiden Operanden in `a + b` ein String, so wird der gesamte Ausdruck als String-Konkatenation ausgeführt.
- Schauen Sie für diese Aufgabe die ASCII-Zeichen im Internet nach. Für die Klausur wird die ASCII-Tabelle angegeben, falls Sie sie benötigen.

Aufgabe 7-2 *Terminierung von Schleifen*

Welche der folgenden Schleifen terminieren, welche terminieren nicht? Begründen Sie Ihre Antworten kurz.

(a)
```
int i = 0;
int j = 0;
do {
    j = j++;
    i = j + i;
} while(i < 200);
```

(b)
```
double i = 0.5;
double j = 0;
while(j < 1) {
    j += i;
    i *= 0.5;
}
```

(c)
```
char i = 'a';
short j = 0;
while (i != j) {
    i++;
    j = (short) (i+i);
}
```

(d)
```
long i = 26L;
long j = 24L;
for (long x = 0L; x < 1000L; x++) {
    i = i / 12 + 23 * (--x);
    j = (x--) + j + 5;
}
```

Lösungsvorschlag:

(a) Die Schleife terminiert nicht. `j++` hat als Rückgabewert 0. Die rechte Seite wird dann um 1 inkrementiert und erst dann wird die linke Seite `j` mit dem alten Wert 0 überschrieben. Damit bleibt `i = j = 0` unverändert in jedem Durchlauf.

(b) Terminiert. Der Wert `j` wächst und nähert sich immer weiter der 1 an. Durch die Rechnerarithmetik wird dies irgendwann zu 1 gerundet.

(c) Terminiert. `i` durchläuft alle `chars`. Irgendwann erreicht es dann auch `'NUL' = 0` mit `valueOf('NUL') = 0`, sodass `i+i` auch 0 ergibt. Dies führt zum Schleifenabbruch.

(d) Terminiert. `x` wird in jedem Schleifendurchlauf um 1 erhöht, aber um 2 verringert. Irgendwann erreicht es `INT_MIN` und der Überlauf sorgt dafür, dass es positiv und `x < 1000` falsch wird.

Aufgabe 7-3 *Kontrollstrukturen*

Im Folgenden sind die Programmausschnitte (a), (b) und (c) gegeben. Betrachten Sie alle vorkommenden Variablen als deklariert und initialisiert.
Überlegen Sie, was in den Schleifen passiert. Wie können Sie den gleichen Effekt ohne Verwendung von Wiederholungsanweisungen erzielen? Geben Sie ggfs. ein entsprechendes Code-Fragment an oder begründen Sie, warum dies nicht möglich ist.

(a)
```
int i = 0;
int j = 42;
while(i < j | i > j){
  i++;
  j--;
}
```

(b)
```
int i = 0;
int j = 42;
while(i < j | i > j);{
  i++;
  j--;
}
```

(c)
```
int i = 0;
int j = 42;
while(i < j | i > j)
  i++;
  j--;
```

Lösungsvorschlag:

Für (a): Der Wert von `i` wird solange erhöht und der Wert von `j` solange verringert, bis sie sich in der Mitte treffen. Derselbe Effekt lässt sich mit der Anweisung `i = j/2; j = i;` erzielen.

Für (b): Die Schleife terminiert nicht, da sie eine leere Anweisung enthält (Semikolon). Code also nicht möglich (oder durch Rekursion).

Für (c): `i` wird 42-mal inkrementiert. Danach wird einmal `j` verringert. Also: `i = j; j--;`

(d) Schreiben Sie für die folgende `do-while`-Schleife jeweils eine äquivalente `while`- und `for`- Schleife. `n` sei eine deklarierte `int`-Variable, die mit einem beliebigen Wert größer 0 initialisiert wurde.

```
double x = 0.;
double y = 1.;
int i = 1;
do {
  x = x + y;
  y = y / i;
} while(i++ <= n);
```

Lösungsvorschlag:

```
// while-Schleife
double x = 1.;
double y = 1.;
int i = 1;
while(i++ <= n) {
        x = x + y;
        y = y / i;
}
```

```
// for-Schleife
double x = 1.;
double y = 1.;
int i;
for(i = 1; i <= n; i++) {
        x = x + y;
        y = y / (i+1);
}
```

Aufgabe 7-4 *Switch (optional)*

Geben Sie für drei gegebenen ganzen Zahlen `x`, `y` und `z` diese Zahlen in aufsteigender Reihenfolge an. Das folgende Codefragment sollen Sie dafür als Grundgerüst verwenden. Implementieren Sie an der mit *TODO* markierten Stelle ein `Switch`-Konstrukt, sodass die Ausgabemethode aufgerufen wird und stets einen gültigen Ausdruck liefert. Ändern Sie dabei nicht die Ausgabemethode. Verwenden Sie keine `if`-Verzweigung.
Sie dürfen die Datei `Switch.java` zum Testen verwenden. Schreiben Sie dort Ihre Lösung an die betreffende Stelle und geben Sie die Datei ab.

```
public static void sortiere(int x, int y, int z){
  int n = 0;
  n += (x > y ? 1 : 0);
  n += (x > z ? 2 : 0);
  n += (y > z ? 4 : 0);
  // TODO
}

public static void ausgabe(int a, int b, int c) {
  if(a > b || a > c || b > c)
    System.out.println("Da stimmt etwas nicht!");
  System.out.println("Es gilt: " + a + " <= " + b + " <= " + c);
}
```

Lösungsvorschlag:

```
switch(n){
  case 1:   ausgabe(y,x,z); break;
  //case 2: nicht erreichbar!
  case 3:   ausgabe(y,z,x); break;
  case 4:   ausgabe(x,z,y); break;
  //case 5:   nicht erreichbar!
  case 6:   ausgabe(z,x,y); break;
  case 7:   ausgabe(z,y,x); break;
  default:  ausgabe(x,y,z);
}
```