TECHNISCHE UNIVERSITÄT DARMSTADT

Kapitel 01b: Grundlagen von Java mit FopBot

Karsten Weihe

TECHNISCHE UNIVERSITÄT DARMSTADT

Ein drittes Programm: Rechteck mit Münzen füllen

Zwei Java-Programme haben wir im ersten Kapitel gesehen. Das dritte Programm wird ein sehr einfaches Bild mit Münzen malen.

Konkret soll ein Rechteck von neun Zeilen und zehn Spalten mit Münzen gefüllt werden, eine Münze auf jedem Feld, also insgesamt neunzig Münzen.

Rechteck füllen

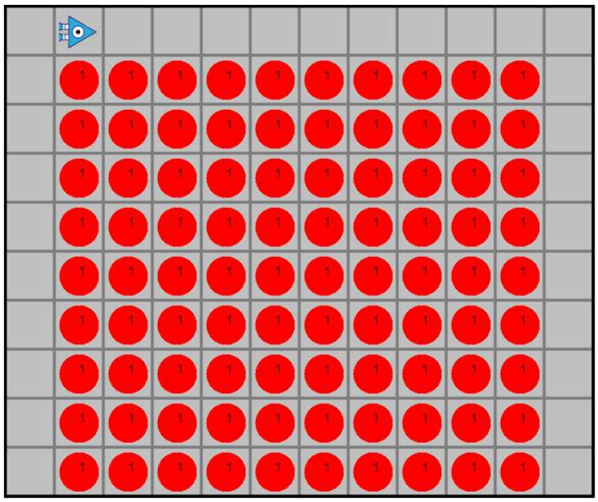
TECHNISCHE UNIVERSITÄT DARMSTADT

```
import fopbot.*;
import static Direction.*;

public class FillRectangle implements Directions {

    public static void main(String[] args) {
        World.setVisible ( true );
        .....
        .....
        .....
    }
}
```

Dieses Programm finden Sie in FillRectangle.java.

Was herauskommen soll

TECHNISCHE UNIVERSITÄT DARMSTADT

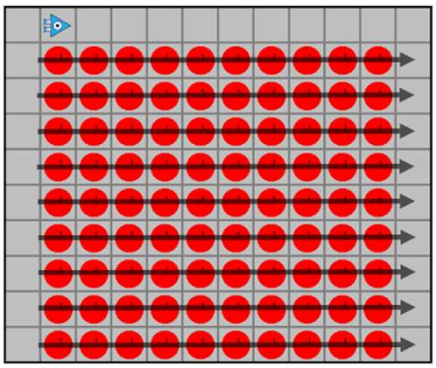


Dieses Bild ist das Ergebnis, das wir erreichen wollen, wobei es nur auf die Münzen ankommt. Wo der Roboter am Ende steht und in welche Richtung er schaut, ist uns egal.

Die Vorgehensweise

TECHNISCHE UNIVERSITÄT DARMSTADT

- Der Robot muss auf jedem Feld genau eine Münze ablegen
- Z.B. in jeder Zeile von links nach rechts
- Problem: Roboter muss zum Anfang der nächsten Zeile navigiert werden

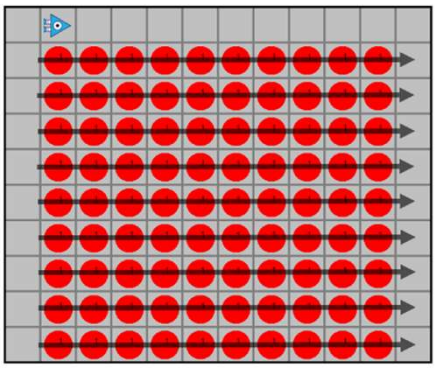


Wir müssen uns klarmachen, wie das Java-Programm überhaupt aussehen könnte, das wir zur Erledigung dieser Aufgabe zu schreiben haben.

Die Vorgehensweise

TECHNISCHE UNIVERSITÄT DARMSTADT

- Der Robot muss auf jedem Feld genau eine Münze ablegen
- Z.B. in jeder Zeile von links nach rechts
- Problem: Roboter muss zum Anfang der nächsten Zeile navigiert werden

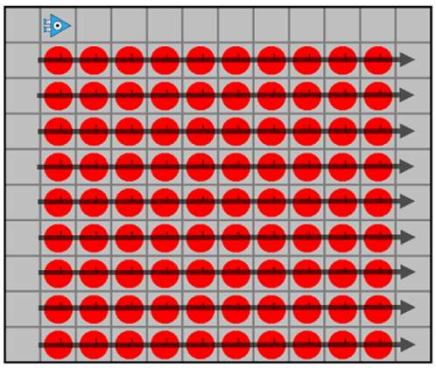


Dazu ist es immer sinnvoll, sich noch einmal auf *verbaler* Ebene klarzumachen, *was* die Aufgabe eigentlich *ist*. Denn bisher hatten wir davon nur ein Bild vor Augen, also nur eine *nonverbale* Vorstellung.

Die Vorgehensweise

TECHNISCHE UNIVERSITÄT DARMSTADT

- Der Robot muss auf jedem Feld genau eine Münze ablegen
- Z.B. in jeder Zeile von links nach rechts
- Problem: Roboter muss zum Anfang der nächsten Zeile navigiert werden



Das Grundschema ist offensichtlich: Entweder gehen wir alle Zeilen jeweils einmal durch, von oben nach unten oder von unten nach oben, und innerhalb jeder Zeile dabei von links nach rechts oder von rechts nach links. Oder wir alle Spalten einmal durch, von links nach rechts oder von rechts nach links, und jede Spalte dabei von oben nach unten oder von unten nach oben.

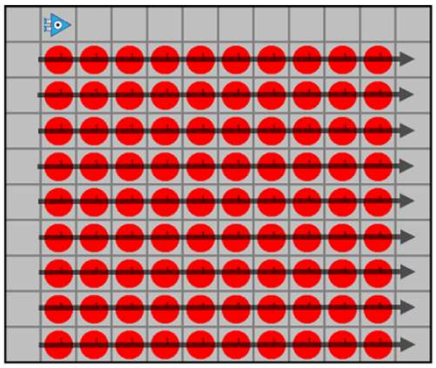
In jedem Fall sieht es verdächtig nach zwei for-Schleifen aus, eine über die Zeilen und eine über die Spalten.

Wie im Bild angedeutet, entscheiden wir uns dafür, jede Zeile von links nach rechts zu durchlaufen. Aber jede andere der genannten Optionen wäre genauso gut gewesen.

Die Vorgehensweise

TECHNISCHE UNIVERSITÄT DARMSTADT

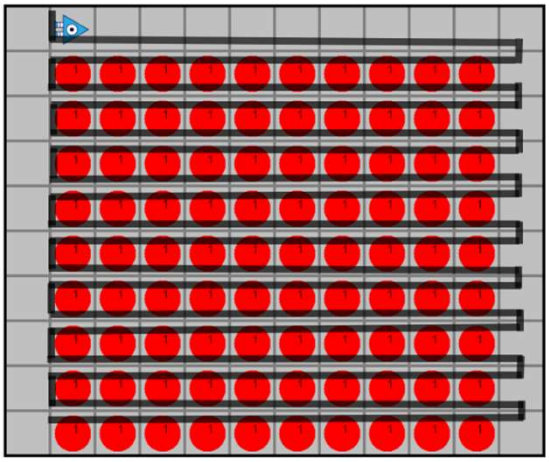
- Der Robot muss auf jedem Feld genau eine Münze ablegen
- Z.B. in jeder Zeile von links nach rechts
- Problem: Roboter muss zum Anfang der nächsten Zeile navigiert werden



Allerdings fehlt offensichtlich noch etwas an der Grundidee. Wenn der Roboter eine Zeile durchlaufen hat, steht er ja an einer völlig falschen Stelle und muss erst an die richtige Stelle für den Durchlauf durch die nächste Zeile kommen.

Die Vorgehensweise

TECHNISCHE UNIVERSITÄT DARMSTADT



Das Java-Programm, das wir entwickeln werden, wird den Roboter auf dem hier gezeigten Pfad systematisch durch das Rechteck führen. An jedem der neunzig Felder, die zum Rechteck gehören, kommt er ein- oder zweimal vorbei, legt aber nur einmal eine Münze ab, nämlich wenn er von links nach rechts läuft.

Wir sehen die prinzipielle Vorgehensweise: Neunmal macht der Roboter dasselbe: Er geht jeweils zehn Schritte von links nach rechts und legt auf jedem Feld eine Münze ab. Dann geht er auf die nächsthöhere Zeile und geht wieder zehn Schritte zurück, um an den Anfang dieser Zeile zu kommen. Beim Zurücklaufen legt er *keine* Münzen ab, da er in dieser Zeile gleich danach auf dem Weg von links nach rechts Münzen ablegen wird.

Wie wir gleich sehen werden, endet der Roboter aufgrund dieser sehr einfachen Schematik eine Zeile höher und schaut wieder nach rechts.

Rechteck füllen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Robot rex = new Robot ( 1, 0, RIGHT, 90 );
.....
.....
.....
.....
```

Als erstes richten wir den Roboter ein. In Anlehnung an das englische Wort rectangle für Rechteck nennen wir ihn rex.

Rechteck füllen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Robot rex = new Robot ( 1, 0, RIGHT, 90 );
.....
.....
.....
.....
```

Und wir richten rex auch gleich mit den benötigten neunzig Münzen ein.

Rechteck füllen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( int i = 0; i < 6; i ++ ) {
   for ( int j = 0; j < 4; j++ ) {
      .........
   }
}
```

i	j
0	0
0	1
0	2
0	3
1	0
1	1
1	2
1	3
2	0
2	1
2	2
2	3
3	0
3	1
3	2
3	3
4	0
4	1
4	2
4	3
5	0
5	1
5	2
5	3

Wir werden zwei for-Schleifen so wie hier ineinander zu schachteln haben, wobei die beiden Zahlen beim Rechteck andere sein werden, nämlich 9 und 10 statt wie hier 6 und 4. Hier stehen kleinere Zahlen, damit die Darstellung rechts noch klein und überschaubar bleibt.

Konkret nehmen wir hier for-Schleifen, aber das Durchlaufprinzip, das Sie rechts sehen, ist absolut dasselbe bei zwei ineinander geschachtelten while-Schleifen oder einer while-Schleife in einer for-Schleife oder einer for-Schleife in einer while-Schleife. Wir haben ja in Kapitel 01a gesehen, dass eine for-Schleife durch ein Codefragment mit while-Schleife ersetzt werden kann.

Rechteck füllen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( int i = 0; i < 6; i ++ ) {
    for ( int j = 0; j < 4; j++ ) {
        ..........
    }
}
```

i	j
0	0
0	1
0	2
0	3
1	0
1	1
1	2
1	3
2	0
2	1
2	2
2	3
3	0
3	1
3	2
3	3
4	0
4	1
4	2
4	3
5	0
5	1
5	2
5	3

Wie Sie rechts sehen, hat die äußere Schleife wie erwartet sechs Durchläufe. Dabei durchläuft i die erwarteten Werte, nämlich 0 bis 5, und vor dem Durchlauf mit i gleich 6 ist die äußere Schleife beendet, da die Fortsetzungsbedingung natürlich nicht mehr erfüllt ist.

Rechteck füllen

TECHNISCHE UNIVERSITÄT DARMSTADT

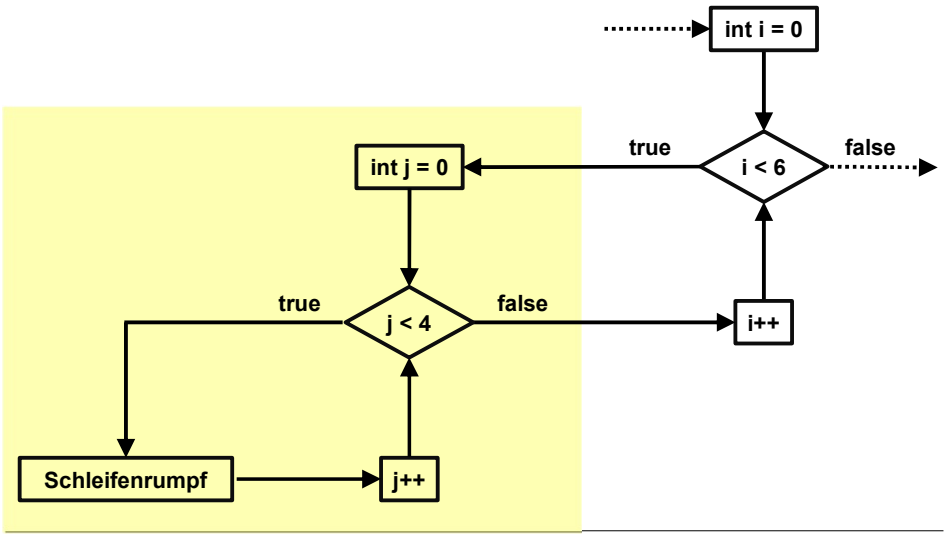
```
for ( int i = 0; i < 6; i ++ ) {
   for ( int j = 0; j < 4; j++ ) {
       .........
   }
}
```

i	j
0	0
0	1
0	2
0	3
1	0
1	1
1	2
1	3
2	0
2	1
2	2
2	3
3	0
3	1
3	2
3	3
4	0
4	1
4	2
4	3
5	0
5	1
5	2
5	3

In jedem Durchlauf durch die äußere Schleife wird einmal die innere Schleife ganz abgearbeitet. Erst wenn die innere Schleife beendet ist, beginnt der nächste Durchlauf durch die äußere Schleife. Die einzelnen Durchläufe durch die äußere Schleife sind hier durch gelbe Striche separiert.

Rechteck füllen

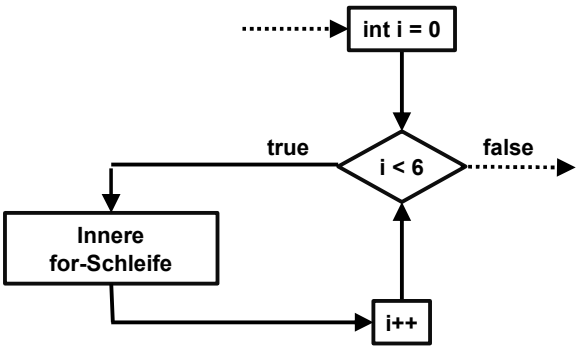
TECHNISCHE UNIVERSITÄT DARMSTADT



Hier sehen Sie das Flussdiagramm dieser zwei ineinander geschachtelten for-Schleifen. Die innere Schleife, gelb unterlegt, ist gerade der Schleifenrumpf der äußeren Schleife. Mit „Schleifenrumpf“ ist hier der der inneren Schleife bezeichnet.

Rechteck füllen

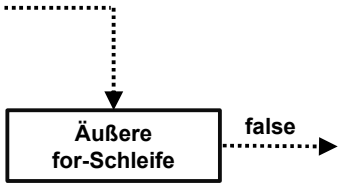
TECHNISCHE UNIVERSITÄT DARMSTADT



Analog zu Abschnitt „Anweisungen abarbeiten“ in Kapitel 01 lässt sich die innere Schleife wieder als eine einzelne Anweisung sehen.

Rechteck füllen

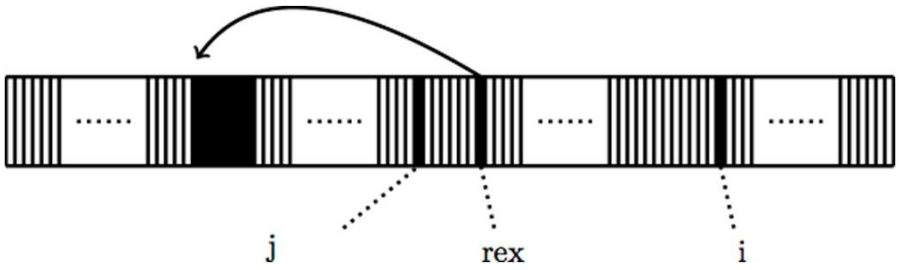
TECHNISCHE UNIVERSITÄT DARMSTADT



Und natürlich können wir einen Schritt weitergehen und das gesamte Konstrukt aus zwei ineinander geschachtelte Schleifen als eine einzelne Anweisung ansehen.

Rechteck füllen

TECHNISCHE UNIVERSITÄT DARMSTADT



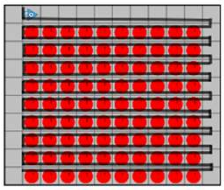
#und so kann man sich das im Computerspeicher vorstellen. Wir hatten schon gesehen, dass bei primitiven Datentypen wie int kein Unterschied gemacht wird zwischen Objekt und Referenz: Der Name i beziehungsweise j spricht unmittelbar die Speicherstelle an, in der die Zahl gespeichert ist.

Die Füllanweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( ... 9 ... ) {
  for ( ... 10 ... ) {
    rex.putCoin();
    rex.move();
  }
  rex.turnLeft();
  rex.move();
  rex.turnLeft();
```

```
  for ( ... 10 ... ) {
    rex.move();
  }
  rex.turnLeft();
  rex.turnLeft();
}
```



Das sind sämtliche Anweisungen, die der Roboter rex ausführt, um das Rechteck mit Münzen unten rechts zu zeichnen. Wir vergessen diese Folie gleich wieder und bauen denselben Code auf den folgenden Folien noch einmal schrittweise auf.

Anweisungen abarbeiten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( int i = 0; i < 9; i++ ) {
  for ( int j = 0; j < 10; j++ ) {
      ..........
  }
  for ( int k = 0; k < 10; k++ ) {
      ..........
  }
}
```

Zu beachten ist aber, dass die Zähler für die Schleifen nicht gleich heißen dürfen. Deshalb nennen wir die Zähler der beiden inneren Schleife nicht i wie bisher, sondern j und k.

Tatsächlich hätten wir die Zähler der beiden inneren Schleifen gleich nennen können, also zum Beispiel beide j. Aber solange wir nicht das Prinzip dahinter gesehen haben, bleiben wir lieber auf der sicheren Seite und benennen *alle* Zähler unterschiedlich.

Anweisungen abarbeiten

TECHNISCHE UNIVERSITÄT DARMSTADT

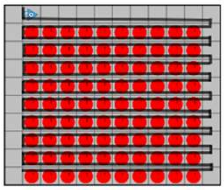
```
for ( int i1 = 0; i1 < 9; i1++ ) {
  for ( int i2 = 0; i2 < 10; i2++ ) {
      ..........
  }
  for ( int i3 = 0; i3 < 10; i3++ ) {
      ..........
  }
}
```

Wir hätten die Zähler beispielsweise auch i1, i2 und i3 nennen können. So sind wir nicht auf die 26 Zeichen des Alphabets beschränkt, sondern könnten Zähler für beliebig viele Schleifen einrichten, wenn wir das wollen.

Die Füllanweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( ... 9 ... ) {
    !! ausfuellen !!
}
```

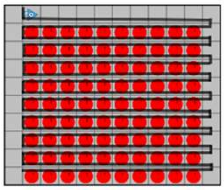


Wie gesagt, werden wir das Programm, das wir eben in Gänze gesehen haben, noch einmal schrittweise aufbauen. Der erste Schritt ist die Erkenntnis, dass neunmal dasselbe passieren wird. Also packen wir alles in eine Schleife, die neunmal durchlaufen wird.

Die Füllanweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( ... 9 ... ) {
  !! ausfuellen !!
}
```



Was in jedem Schleifendurchlauf passieren muss, um die unten rechts skizzierte Durchlaufstrategie zu realisieren, müssen wir uns erst noch überlegen. Im ersten Schritt setzen wir hier nur eine Erinnerungsstütze hinein.

So eine Erinnerungsstütze muss nicht sein. Es ist aber selbst für erfahrene Programmierer sinnvoll, sich eine Zeichenkette zu überlegen, die garantiert so nicht im Quelltext vorkommt, und exakt diese Zeichenkette überall dort einzusetzen, wo noch etwas zu tun ist. Denn so kann man jede solche Stelle mit einem einzigen Suchbefehl finden – selbst wenn das Programm zehntausende Zeilen Code hat und sich über mehrere Dateien verteilt.

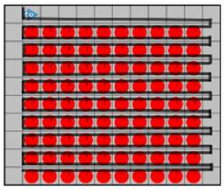
Diese Zeichenkette sollte auch nicht in einen Kommentar gesetzt werden, denn so wie auf dieser Folie – eben *nicht* in einem Kommentar – wird der Compiler uns praktischerweise mit einer Fehlermeldung darauf aufmerksam machen, wenn wir eine Stelle vergessen haben zu füllen.

Die Füllanweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( ... 9 ... ) {
   for ( ... 10 ... ) {
      !! ausfuellen !!
   }
   !! ausfuellen !!
}
```

```
for ( ... 10 ... ) {
   !! ausfuellen !!
}
!! ausfuellen !!
```



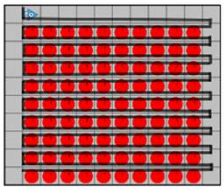
Wir machen jetzt den nächsten Schritt in unseren Überlegungen. Die schließende Klammer der allumfassenden for-Schleife ist nun weit weg vom Kopf der Schleife.

Die Füllanweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( ... 9 ... ) {
  for ( ... 10 ... ) {
    !! ausfuellen !!
  }
  !! ausfuellen !!
```

```
for ( ... 10 ... ) {
  !! ausfuellen !!
}
!! ausfuellen !!
}
```



Wir hatten schon früher gesehen, wie der Kopf einer for-Schleife im Detail eigentlich aussieht und wie wir ihn abgekürzt auf den Folien notieren, eben nur die Anzahl Durchläufe und alles davor und danach in den runden Klammern nur durch drei Punkte angedeutet.

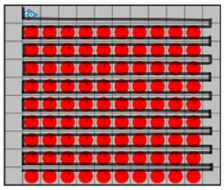
Sie können leicht raten, was die Funktion der einzelnen Schleifen ist: Die äußere Schleife geht Zeile für Zeile durch, die erste innere Schleife geht die momentane Zeile von links nach rechts durch und legt Münzen ab, und die zweite innere Schleife bewegt den Roboter wieder ganz nach links zurück.

Die Füllanweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( ... 9 ... ) {
  for ( ... 10 ... ) {
    rex.putCoin();
    rex.move();
  }
  !! ausfuellen !!
```

```
  for ( ... 10 ... ) {
    !! ausfuellen !!
  }
  !! ausfuellen !!
}
```



Als nächstes überlegen wir uns, was beim Durchlauf einer Zeile von links nach rechts genau passieren soll, also was in der ersten inneren Schleife stehen soll.

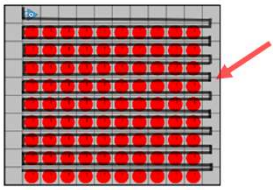
Wir haben schon gesehen, wie man Münzen in einer Zeile oder Spalte nacheinander ablegt. Die Schleife sieht hier genauso aus: Zuerst wird eine Münze abgelegt, dann geht der Roboter einen Schritt vorwärts. Zehnmal insgesamt, da das Rechteck ja zehn Spalten umfassen soll.

Die Füllanweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( ... 9 ... ) {
  for ( ... 10 ... ) {
    rex.putCoin();
    rex.move();
  }
 rex.turnLeft();
 rex.move();
 rex.turnLeft();
```

```
for ( ... 10 ... ) {
  !! ausfuellen !!
 }
 !! ausfuellen !!
}
```



Aber das reicht nicht. Nachdem eine Zeile gefüllt ist, steht der Roboter am Ende dieser Zeile und schaut nach rechts. Zur Füllung der nächsten Zeile von links nach rechts muss er aber am *Anfang* der *nächsten* Zeile stehen und dort nach rechts schauen. In diese Position müssen wir den Roboter erst noch bringen.

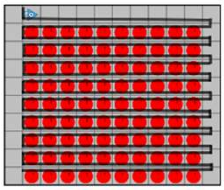
Um an den Anfang der nächsten Zeile zu kommen, müssen wir wenden, also zweimal turnLeft. Nach einem turnLeft schaut der Roboter nach oben. Diese Gelegenheit nutzen wir, um mit einem move zwischen den beiden turnLeft in die nächste Zeile zu kommen.

Die Füllanweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( ... 9 ... ) {
  for ( ... 10 ... ) {
    rex.putCoin();
    rex.move();
  }
  rex.turnLeft();
  rex.move();
  rex.turnLeft();
```

```
for ( ... 10 ... ) {
  rex.move();
}
!! ausfuellen !!
}
```

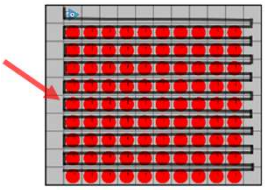


Jetzt ist der Roboter schon auf der richtigen Zeile, aber am rechten Ende und schaut nach links. Um an den Anfang der Zeile zu kommen, geht der Roboter erst einmal in einer weiteren Schleife um zehn Spalten nach links. Hier setzt er keine Münze ab, denn wir haben entschieden, dass Münzen im Durchlauf von links nach rechts abgesetzt werden und daher keine Münze im Durchlauf von rechts nach links gesetzt werden darf, da ja auf jedem Feld nur *eine* Münze abgesetzt werden soll.

Die Füllanweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

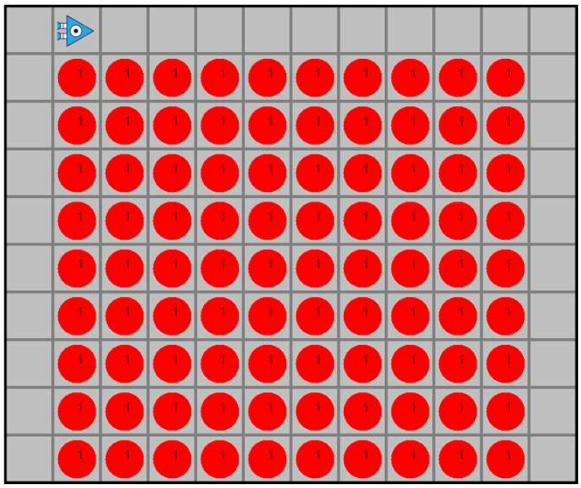
```
for ( ... 9 ... ) {
  for ( ... 10 ... ) {
    rex.putCoin();
    rex.move();
  }
  rex.turnLeft();
  rex.move();
  rex.turnLeft();
  for ( ... 10 ... ) {
    rex.move();
  }
  rex.turnLeft();
  rex.turnLeft();
}
```



Jetzt ist der Roboter an der richtigen Position, schaut aber nach links. Also müssen wir ihn noch einmal um 180 Grad wenden lassen. Und dann ist er für das Füllen der nächsten Zeile bereit, mehr ist in einem einzelnen Durchlauf der äußeren Schleife nicht zu tun, der nächste Durchlauf eine Zeile höher kann beginnen.

Noch einmal das Ergebnis

TECHNISCHE UNIVERSITÄT DARMSTADT



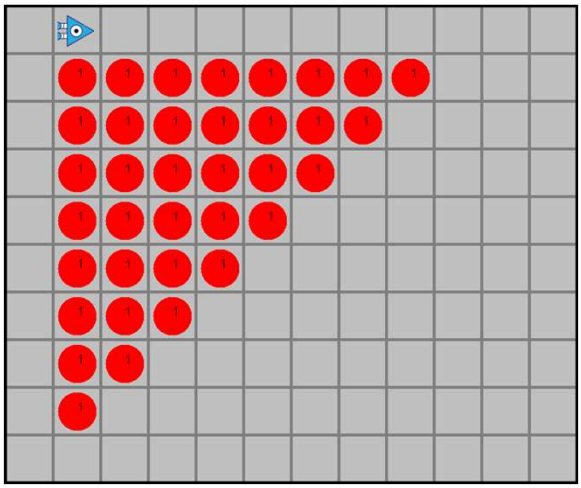
Hier noch einmal das Ergebnis in voller Größe. Der Roboter ist zum Füllen einer weiteren, der zehnten Zeile bereit, weil er unnötigerweise im letzten Schleifendurchlauf noch einmal so positioniert wurde. Macht nichts, es gibt ja wie gesagt keine Vorgabe, wo der Roboter am Ende stehen und in welche Richtung er schauen soll.

TECHNISCHE UNIVERSITÄT DARMSTADT

Dreieck statt Rechteck füllen

Zu Zwecken der Illustration variieren wir unsere Lösung für die Aufgabe, ein Rechteck mit Münzen zu füllen.

TECHNISCHE UNIVERSITÄT DARMSTADT



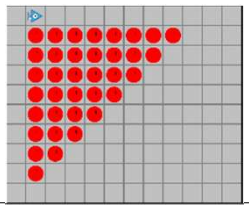
Das Ergebnis soll so aussehen wie auf diesem Bild. Wieder kommt es uns nur auf die Münzen an, nicht auf die Endposition des ausführenden Roboters. Der Roboter startet wieder auf derselben Position wie eben, also zweite Spalte, erste Zeile. In dieser Zeile ist keine Münze abzulegen.

Wir hätten den Roboter daher auch eine Zeile höher platzieren können, aber es ist ganz instruktiv zu sehen, wie es aussieht, wenn wir die for-Schleife mit 0 Münzen starten.

Dreieck füllen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( int i = 0; i < 9; i++ ) {
  for ( int j = 0; j < i; j++  ) {
    !! ausfuellen !!
  }
  !! ausfuellen !!
  for ( int k = 0; k < i; k++ ... ) {
    !! ausfuellen !!
   }
   !! ausfuellen !!
}
```

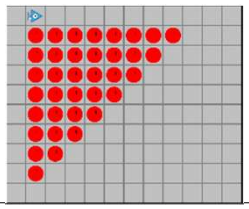


Das Programm für das Ausfüllen eines Rechtecks muss nur an zwei Stellen überhaupt geändert werden, um statt dessen dieses Dreieck zu erzeugen. Diese beiden Stellen hatten wir schon zu einer frühen Phase in der Entwicklung des Programm fertiggestellt, als für mehrere Teile des Programms noch ein Platzhalter „!! ausfuellen !!“ stand. Auf dieser Phase setzen wir auf.

Dreieck füllen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( int i = 0; i < 9; i++ ) {
  for ( int j = 0; j < i; j++  ) {
    !! ausfuellen !!!
  }
  !! ausfuellen !!
  for ( int k = 0; k < i; k++ ... ) {
    !! ausfuellen !!
  }
  !! ausfuellen !!
}
```



Das sind die beiden besagten Stellen, an denen das Programm zu modifizieren ist. Die Modifikation ist offensichtlich zweimal praktisch dieselbe.

Machen Sie sich klar: Alles, was auf dieser Folie noch nicht ausgefüllt ist, ist absolut identisch zu dem Quelltext für das Zeichnen eines Rechtecks weiter vorn. Die beiden Quelltexte unterscheiden sich nur darin, wie viele Schritte der Roboter in jeder Zeile nach rechts geht (und dann wieder nach links zurückgehen muss), also in der Fortsetzungsbedingung der beiden inneren for-Schleifen.

Dreieck füllen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( int i = 0; i < 6; i ++ ) {
   for ( int j = 0; j < i; j++ ) {
      .........
   }
}
```

i	j
1	0
2	0
2	1
3	0
3	1
3	2
4	0
4	1
4	2
4	3

i	j
5	0
5	1
5	2
3	3
5	4

Eben beim Rechteck hatten wir uns das ähnlich schematisch wie jetzt auf dieser Folie angeschaut, auch wieder mit kleineren Zahlen als im Quelltext. Auf dieser Folie sehen Sie die Anpassung dieser schematischen Darstellung auf das Dreieck. Der Unterschied ist, dass nun die innere Schleife bei jedem Mal eine andere Zahl von Durchläufen hat.

Für i gleich 0 gibt es keinen einzigen Durchlauf durch die innere Schleife. Das ist für unsere Dreieckszeichnung auch genau richtig, denn auf der Startzeile des Roboters soll ja auch keine einzige Münze abgelegt werden.

TECHNISCHE UNIVERSITÄT DARMSTADT

Rechteck mit Münzen füllen mit Hilfe von Variablen

Wir schreiben das Programm zum Füllen eines Rechtecks mit Münzen noch einmal leicht um.

Rechteck füllen mit Variablen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import fopbot.*;
import static Direction.*;
public class FillRectangleVariables implements Directions {
   public static void main(String[] args)  {
      World.setSize ( 12, 10 );
      World.setVisible ( true );
      Robot rex = new Robot ( 1, 0, RIGHT, 90 );
      ..........
      ..........
      ..........
   }
}
```

Diese Variation des Programms finden Sie in der Datei FillRectangleVariables.java.

Rechteck füllen mit Variablen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import fopbot.*;
import static Direction.*;
public class FillRectangleVariables implements Directions {
   public static void main(String[] args)  {
      World.setSize ( 12, 10 );
      World.setVisible ( true );
      Robot rex = new Robot ( 1, 0, RIGHT, 90 );
      .........
      .........
      .........
   }
}
```

Wir richten einen Roboter wieder mit denselben Parameterwerten ein und nennen ihn wieder rex.

Rechteck füllen mit Variablen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Robot rex = new Robot ( 1, 0, RIGHT, 90 );
int numberOfRows    =  9;
int numberOfColumns = 10;
.....
.....
.....
```

Zusätzlich zum Roboter rex richten wir diesmal noch zwei weitere Variable ein.

Rechteck füllen mit Variablen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Robot rex = new Robot ( 1, 0, RIGHT, 90 );
int numberOfRows      =  9;
int numberOfColumns  = 10;
.....
.....
.....
```

Diese sind nicht wie alle bisherigen Variablen von Klasse Robot, sondern sie sind vom *primitiven Datentyp* int, den wir schon an verschiedenen Stellen in Kapitel 01a gesehen hatten.

Das heißt, jede dieser beiden Variablen kann eine ganze Zahl speichern: eine positive ganze Zahl, eine negative ganze Zahl oder natürlich auch die 0.

Rechteck füllen mit Variablen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Robot rex = new Robot ( 1, 0, RIGHT, 90 );
int numberOfRows    =  9;
int numberOfColumns = 10;
.........
.........
.........
```

Die beiden Variablen nennen wir numberOfRows und numberOfColumns, denn genau diese beiden Anzahlen sollen die beiden Variablen enthalten: die Anzahl der Zeilen und Spalten, die das Rechteck umfasst.

***Erinnerung* an Kapitel 01a, Abschnitt zu Identifiern: Es ist Konvention in Java, dass der erste Buchstabe eines Variablennamens klein geschrieben wird und ein neues Wort innerhalb eines Variablennamens durch großen ersten Buchstaben angezeigt wird.**

Rechteck füllen mit Variablen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Robot rex = new Robot ( 1, 0, RIGHT, 90 );
int numberOfRows    =  9;
int numberOfColumns = 10;
.....
.....
.....
```

Wir wollten neun Zeilen und zehn Spalten füllen, also setzen wir die beiden Variablen auf den Wert 9 beziehungsweise 10.

Das Gleichheitszeichen ist in Java der Zuweisungsoperator, das heißt, der Variable auf der linken Seite des Gleichheitszeichens wird der Wert auf der rechten Seite zugewiesen. Die Variable numberOfRows speichert also jetzt den Wert 9 und die Variable numberOfColumns den Wert 10.

Rechteck füllen mit Variablen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Robot rex = new Robot ( 1, 0, RIGHT, 90 );
int numberOfRows      =   9;
int numberOfColumns  = 10;
.....
.....
.....
```

Wir sehen: Egal von welchem Typ, Variable werden in Java *immer* so deklariert, dass zuerst der Typ der Variable angegeben wird und danach ihr Name.

Rechteck füllen mit Variablen

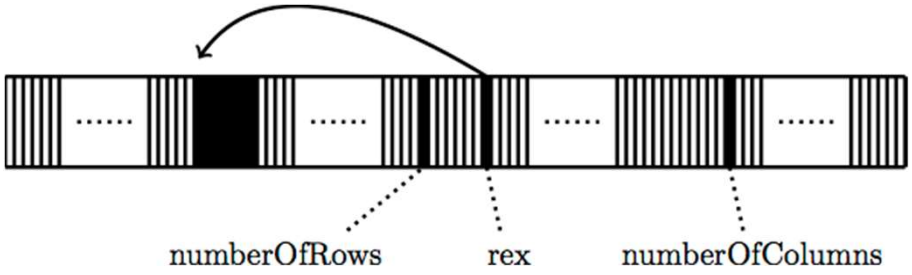
TECHNISCHE UNIVERSITÄT DARMSTADT

```
Robot rex = new Robot ( 1, 0, RIGHT, 90 );
int numberOfRows    =  9;
int numberOfColumns = 10;
.....
.....
.....
```

Aber bei der *Initialisierung* einer Variable bestehen doch erhebliche Unterschiede: Bei *primitiven* Datentypen wie int reicht es einfach, den Wert auf die rechte Seite des Gleichheitszeichen zu setzen. Robot ist hingegen eine *Klasse*. Klassen sind komplexer und müssen auch komplexer initialisiert werden.

Rechteck füllen mit Variablen

TECHNISCHE UNIVERSITÄT DARMSTADT




```
Robot rex = new Robot ( 1, 2, RIGHT, 90 );
int numberOfRows    =  9;
int numberOfColumns = 10;
```

Den Unterschied zwischen Klassen und primitiven Datentypen hatten wir schon gesehen, der ist in diesem Beispiel natürlich genauso. Dieses Bild erklärt, warum bei rex das eigentliche Objekt mit Operator new erzeugt werden muss und bei den beiden Variablen vom Typ int nicht.

Wieder die Füllanweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( ... numberOfRows ... ) {
   for ( ... numberOfColumns ... ) {
      rex.putCoin();
      rex.move();
    }
    rex.turnLeft();
    rex.move();
    rex.turnLeft();
    for ( ... numberOfColumns ... ) {
      rex.move();
    }
    rex.turnLeft();
    rex.turnLeft();
}
```

Noch einmal dieselben Anweisungen zum Füllen des Rechtecks wie im Programm FillRectangle.

Wieder die Füllanweisungen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( ... numberOfRows ... ) {
   for ( ... numberOfColumns ... ) {
      rex.putCoin();
      rex.move();
   }
   rex.turnLeft();
   rex.move();
   rex.turnLeft();
   for ( ... numberOfColumns ... ) {
      rex.move();
   }
   rex.turnLeft();
   rex.turnLeft();
}
```

Anstelle der Zahlenwerte stehen jetzt die Variablennamen in den Schleifen. Das Ergebnis ist natürlich exakt dasselbe. Aber durch die sprechenden Namen, numberOfRows und numberOfColumns, sollte das Programm leichter verständlich sein.

Bei einem so kleinen Programm mag leichtere Verständlichkeit noch keine Rolle spielen. Aber bei Programmen mit mehreren Millionen Zeilen ist man beim Lesen und Verstehen des Programms schon froh, wenn man solche sprechenden Namen anstelle von irgendwelchen Zahlenwerten findet.

Und stellen Sie sich vor, die Anzahl der Zeilen ist gleich der Anzahl der Spalten. Wenn Sie nur die Anzahl überall als Zahlenwert hinschreiben, müssen Sie sich auch überall bei der Arbeit mit dem Quelltext erst einmal überlegen, ob die Anzahl der Zeilen oder die Anzahl der Spalten gemeint ist. Wenn Sie nur eins von beidem ändern müssen und das andere so belassen wollen, sind schwer zu findende Fehler in einem größeren

Programm kaum vermeidbar.

TECHNISCHE UNIVERSITÄT DARMSTADT

Weiter:

Anweisungen abarbeiten

Oracle Java Tutorials: Control Flow Statements

Wir hatten in Kapitel 01a, Abschnitt „Allgemein: Programmablauf“, schon damit begonnen, uns allgemein anzusehen, wie Anweisungen abgearbeitet werden. Mit dem seither gewonnenen zusätzlichen Wissen und Verständnis können wir damit nun fortfahren.

Anweisungen abarbeiten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
while ( true ) {
   if ( sun.isShining() ) {
     break;
  }
   if ( moon.IsWaning() ) {
     continue;
   }
   myRobot.move();
}
myRobot.putCoin();
```

1.
2. Falls die Sonne scheint, springe zu 6
3. Falls der Mond abnehmend ist, springe zu 1
4. Führe myRobot.move() aus
5. Springe zu 1
6. Führe myRobot.putCoin() aus

Auf dieser Folie sehen wir drei weitere Schlüsselwörter von Java, die wir bisher noch nicht diskutiert haben: if, break und continue. Die if-Verzweigung ist ein ganz neues programmiersprachliches Konstrukt, das nichts mit Schleifen zu tun hat, während break und continue zum Thema Schleifen dazugehören.

Anweisungen abarbeiten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
while ( true ) {
   if ( sun.isShining() ) {
     break;
   }
   if ( moon.IsWaning() ) {
     continue;
   }
   myRobot.move();
}
myRobot.putCoin();
```

1.
2. Falls die Sonne scheint, springe zu 6
3. Falls der Mond abnehmend ist, springe zu 1
4. Führe myRobot.move() aus
5. Springe zu 1
6. Führe myRobot.putCoin() aus

Das Wort true ist nominell kein Schlüsselwort, aber ebenfalls reserviert. Wie wir bald sehen werden, stehen die Wörter true und false für die Wahrheitswerte wahr und falsch. Das Wort true als Fortsetzungsbedingung bedeutet logischerweise, dass die Fortsetzungsbedingung immer true, also wahr ist. Im alternativen Modell braucht man daher an der entsprechenden Stelle keinen bedingten Sprung zur nächsten Anweisung nach der Schleife, denn die Bedingung für den Sprung – dass eben die Fortsetzungsbedingung nicht erfüllt ist – ist ja nie gegeben.

Damit hätten wir im Prinzip eine Endlosschleife, also eine Schleife, aus der der Prozess niemals herausspringt. Wir werden aber gleich sehen, dass der Prozess aus der Schleife auf dieser Folie durchaus herausspringen kann.

Anweisungen abarbeiten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
while ( true ) {
   if ( sun.isShining() ) {
     break;
  }
   if ( moon.IsWaning() ) {
     continue;
   }
   myRobot.move();
}
myRobot.putCoin();
```

1.
2. Falls die Sonne scheint, springe zu 6
3. Falls der Mond abnehmend ist, springe zu 1
4. Führe myRobot.move() aus
5. Springe zu 1
6. Führe myRobot.putCoin() aus

Die Variablen sun und moon sollen von Klassen sein, deren Objekte Himmelskörper repräsentieren, und die wir hier als gegeben voraussetzen. Sie sind *nicht* Teil von FopBot, wir müssten sie eigentlich selbst definieren, aber das unterlassen wir hier.

Das werden wir häufiger machen: irgendwelche Funktionalität einfach als gegeben voraussetzen, ohne sie zu implementieren. Mit einer solchen Situation umzugehen, ist eine grundlegende Kompetenz in der Informatik.

Anweisungen abarbeiten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
while ( true ) {
   if ( sun.isShining() ) {
      break;
   }
   if ( moon.IsWaning() ) {
      continue;
   }
   myRobot.move();
}
myRobot.putCoin();
```

1.
2. Falls die Sonne scheint, springe zu 6
3. Falls der Mond abnehmend ist, springe zu 1
4. Führe myRobot.move() aus
5. Springe zu 1
6. Führe myRobot.putCoin() aus

Die bedingten Sprünge im alternativen Modell bilden ein Konstrukt, das es auch in Java gibt, die if-Verzweigung: Falls die Bedingung in runden Klammern wahr ist, werden die Anweisungen in geschweiften Klammern ausgeführt; ansonsten wird der Block in geschweiften Klammern ganz übersprungen, das heißt, es geht mit der nächsten Anweisung danach weiter.

Anweisungen abarbeiten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
while ( true ) {
   if ( sun.isShining() ) {
     break;
   }
   if ( moon.IsWaning() ) {
     continue;
   }
   myRobot.move();
}
myRobot.putCoin();
```

1.
2. Falls die Sonne scheint, springe zu 6
3. Falls der Mond abnehmend ist, springe zu 1
4. Führe myRobot.move() aus
5. Springe zu 1
6. Führe myRobot.putCoin() aus

Mit der break-Anweisung wird die Schleife verlassen, das heißt, die nächste Anweisung nach der Schleife wird ausgeführt. Im Falle von mehreren ineinander geschachtelten Schleifen wird durch das break die innerste Schleife verlassen.

Anweisungen abarbeiten

TECHNISCHE UNIVERSITÄT DARMSTADT

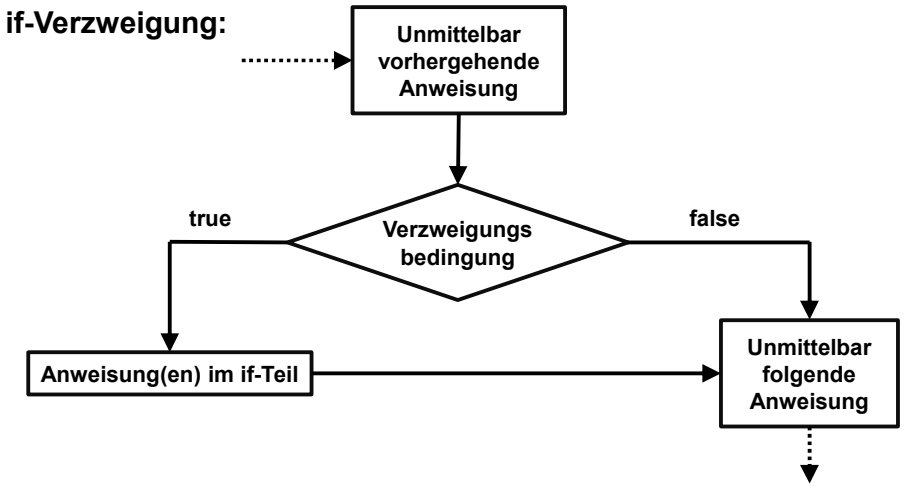
```
while ( true ) {
   if ( sun.isShining() ) {
      break;
   }
   if ( moon.IsWaning() ) {
      continue;
   }
   myRobot.move();
}
myRobot.putCoin();
```

1.
2. Falls die Sonne scheint, springe zu 6
3. Falls der Mond abnehmend ist, springe zu 1
4. Führe myRobot.move() aus
5. Springe zu 1
6. Führe myRobot.putCoin() aus

Im Gegensatz zur break-Anweisung beendet die continue-Anweisung nicht die Schleife als Ganzes, sondern nur den momentanen *Durchlauf*. Das heißt, der gesamte Rest des Schleifenrumpfs wird übersprungen, und als nächstes wird wieder die Fortsetzungsbedingung getestet (die in diesem Fall true ist).

Anweisungen abarbeiten

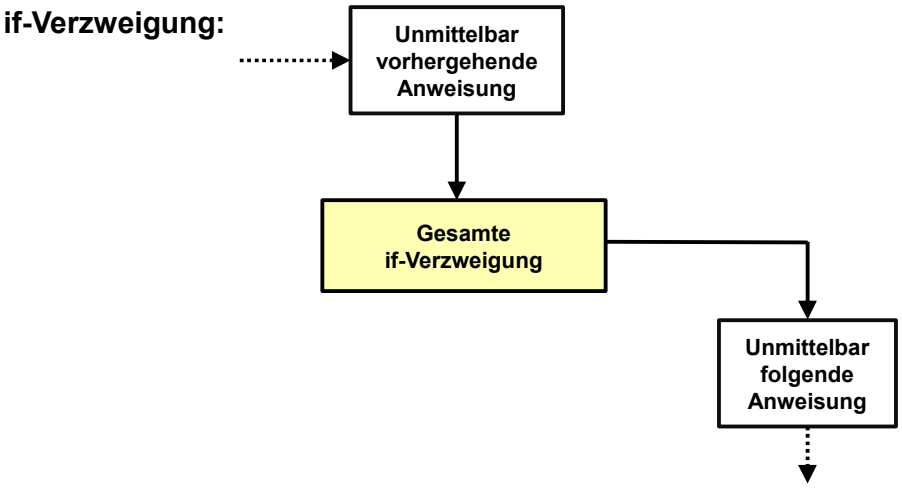
TECHNISCHE UNIVERSITÄT DARMSTADT



Hier sehen Sie das Flussdiagramm der if-Verzweigung: Wenn die Bedingung true ist, dann gibt es einen „Schlenker“ über die zugehörigen Anweisungen, sonst werden diese einfach übersprungen.

Anweisungen abarbeiten

TECHNISCHE UNIVERSITÄT DARMSTADT



Wie bei Schleifen, lässt sich auch eine if-Verzweigung als einzelne Anweisung ansehen.

Anweisungen abarbeiten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
myRobot.move();
if ( myRobot.getY() < 10 ) {
   myRobot.turnLeft();
}
else {
   myRobot.putCoin();
}
myRobot.move();
```

1. Führe myRobot.move() aus
2. Falls nicht myRobot.getY() < 10, springe zu 5
3. Führe myRobot.turnLeft() aus
4. Springe zu 6
5. Führe myRobot.putCoin() aus
6. Führe myRobot.move() aus

Wir schauen uns die if-Verzweigung genauer an.

Anweisungen abarbeiten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
myRobot.move();
if ( myRobot.getY() < 10 ) {
   myRobot.turnLeft();
}
else {
   myRobot.putCoin();
}
myRobot.move();
```

1. Führe myRobot.move() aus
2. Falls nicht myRobot.getY() < 10, springe zu 5
3. Führe myRobot.turnLeft() aus
4. Springe zu 6
5. Führe myRobot.putCoin() aus
6. Führe myRobot.move() aus

So wie hier farblich unterlegt, haben wir die if-Verzweigung in Java und im alternativen Modell auf der letzten Folie schon gesehen.

Anweisungen abarbeiten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
myRobot.move();
if ( myRobot.getY() < 10 ) {
   myRobot.turnLeft();
}
else {
   myRobot.putCoin();
}
myRobot.move();
```

1. Führe myRobot.move() aus
2. Falls nicht myRobot.getY() < 10, springe zu 5
3. Führe myRobot.turnLeft() aus
4. Springe zu 6
5. Führe myRobot.putCoin() aus
6. Führe myRobot.move() aus

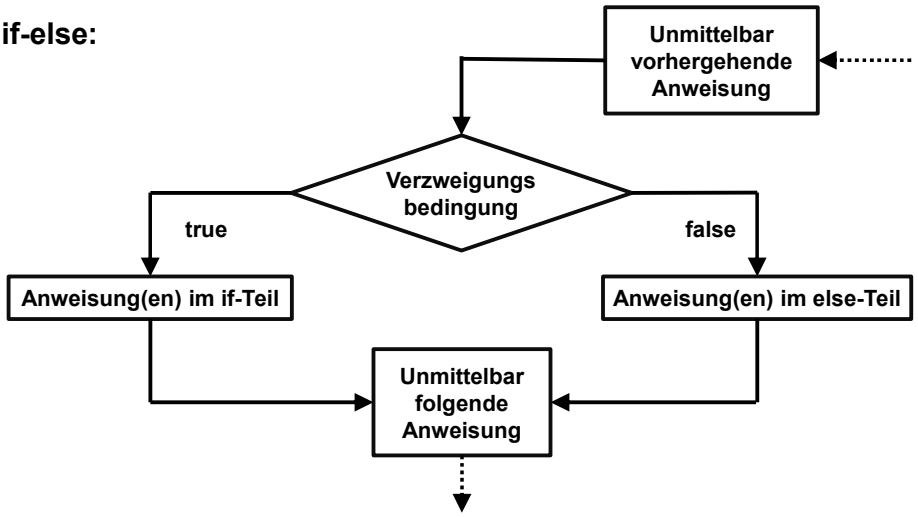
Das ist jetzt neu: Falls die Fortsetzungsbedingung nicht erfüllt ist, wird alternativ ein zweiter Block von Anweisungen ausgeführt. Das Schlüsselwort else trennt diese beiden Blöcke.

Die if-Verzweigung gibt es also in zwei Formen: zum einen mit else und zweitem Block so wie auf dieser Folie, zum anderen ohne wie in den vorangegangenen Beispielen.

Anweisungen abarbeiten

TECHNISCHE UNIVERSITÄT DARMSTADT

if-else:

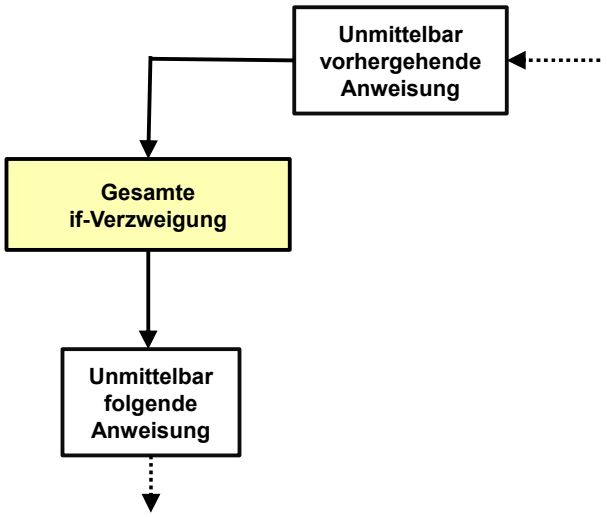


So sieht also das Flussdiagramm einer if-else-Verzweigung aus. Die if-else-Verzweigung realisiert ein Entweder-Oder: *entweder* werden die Anweisungen im if-Teil ausgeführt, *oder* die Anweisungen im else-Teil, aber im selben Durchlauf durch die if-else-Verzweigung nicht beide (in unterschiedlichen Durchläufen durch dieselbe if-else-Verzweigung kann natürlich schon einmal der if-Teil, das andere Mal der else-Teil oder umgekehrt durchlaufen werden, je nachdem, ob die Verzweigungsbedingung jeweils true oder false ist).

Anweisungen abarbeiten

TECHNISCHE UNIVERSITÄT DARMSTADT

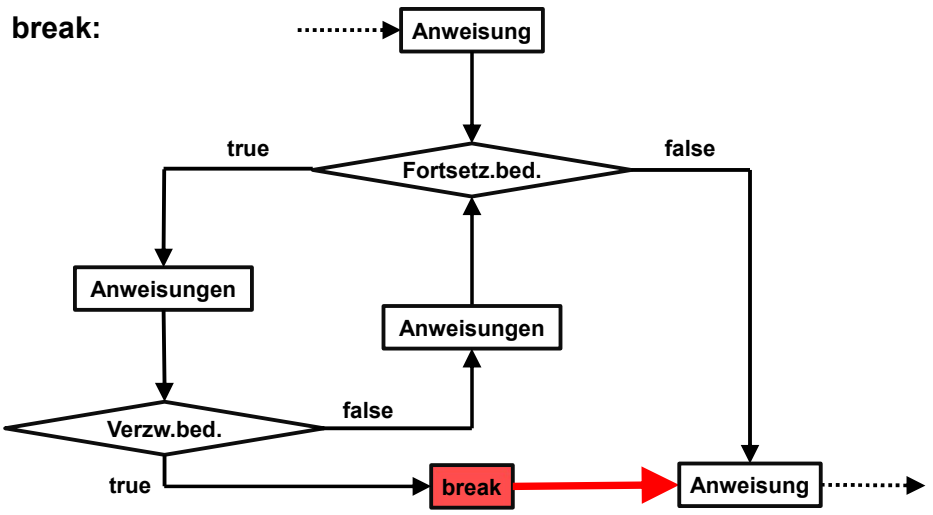
if-else:



Mit und ohne else-Teil sieht das abstrahierte Bild völlig gleich aus.

Anweisungen abarbeiten

TECHNISCHE UNIVERSITÄT DARMSTADT

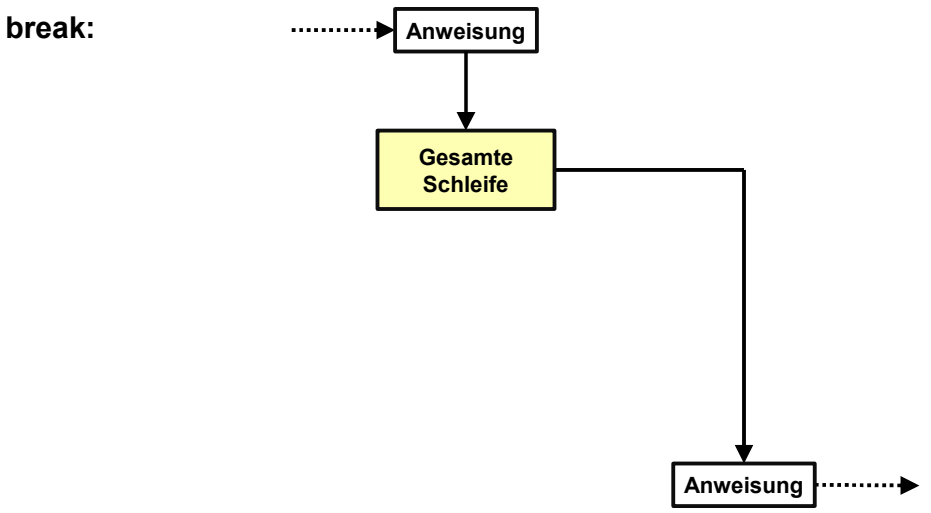


Läuft der Prozess in eine break-Anweisung in einer Schleife, dann wird die Schleife beendet. Sind mehrere Schleifen ineinander geschachtelt, dann wird nur die innerste Schleife beendet unter allen, die das break umschließen. Das können, while-Schleifen, do-while-Schleifen und for-Schleifen sein, egal.

Auf dieser Folie sehen Sie das Flussdiagramm einer typischen Anwendung der break-Anweisung. Im Regelfall macht break nur Sinn im Rahmen einer if-Verzweigung so wie hier, denn ansonsten könnte man den Rest des Schleifenrumpfes ja auch gleich aus dem Programm löschen, alles hinter break würde sowieso nie ausgeführt.

Anweisungen abarbeiten

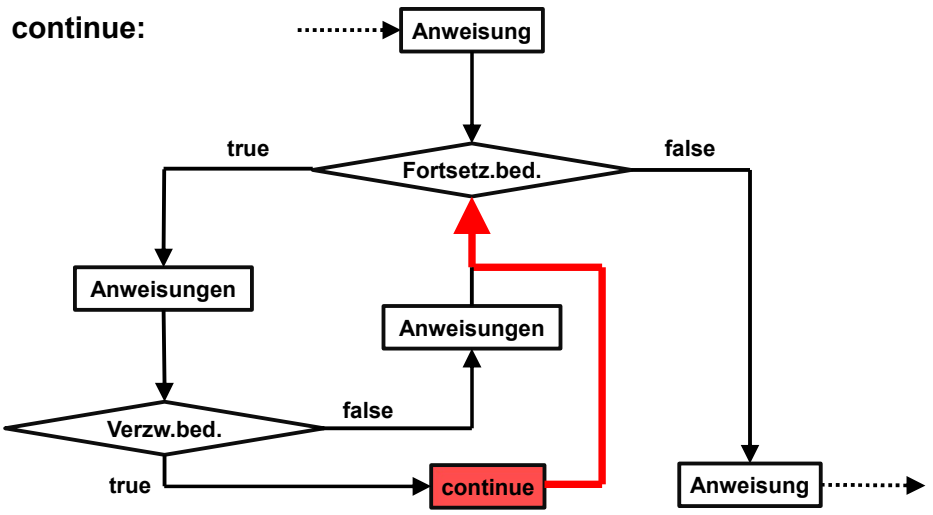
TECHNISCHE UNIVERSITÄT DARMSTADT



Die break-Anweisung in der Schleife ändert nichts daran, dass die gesamte Schleife genau einen Eintrittspunkt und genau einen Austrittspunkt hat.

Anweisungen abarbeiten

TECHNISCHE UNIVERSITÄT DARMSTADT

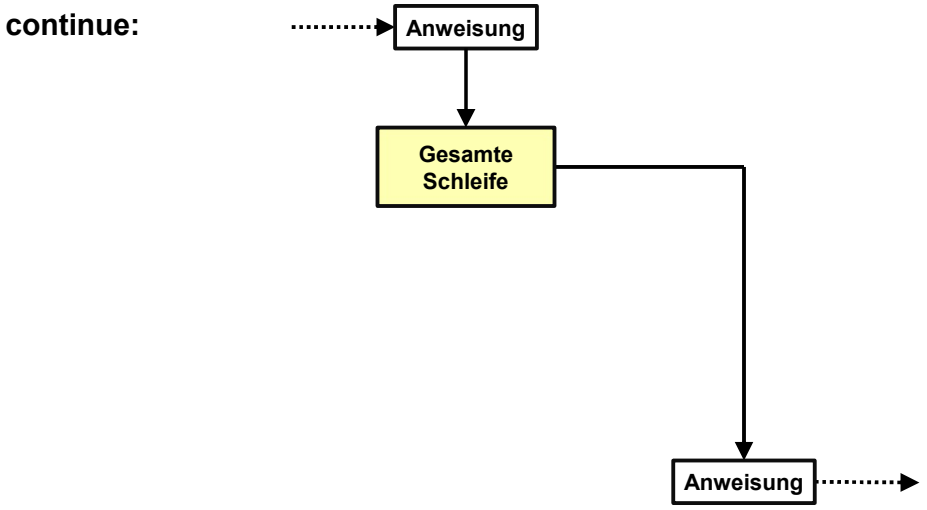


Die continue-Anweisung ist völlig analog zur break-Anweisung. Der Unterschied liegt im Effekt: Es wird nicht die innerste umfassende Schleife ganz abgebrochen, sondern nur der momentane *Durchlauf* durch die innerste umfassende Schleife. Das heißt, der Prozess springt nicht wie bei break vorwärts zur nächsten Anweisung nach der Schleife, sondern zurück an ihren Anfang. Bei der while- und for-Schleife ist das der Test der Fortsetzungsbedingung, bei der do-while-Schleife der Anfang des Schleifenrumpfs.

Die beiden Wörter break und continue sind also recht gut gewählt: break bricht die Schleife ab, continue setzt sie mit dem nächsten Durchlauf fort.

Anweisungen abarbeiten

TECHNISCHE UNIVERSITÄT DARMSTADT



Auch eine continue-Anweisung in der Schleife ändert nichts am abstrahierten Gesamtbild.

Anweisungen abarbeiten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
while ( ..... ) {
   <<Einzelne Anweisung 1>>
}

for ( .....; .....; ..... ) {
   << Einzelne Anweisung 2>>
}

if ( ..... ) {
   <<Einzelne Anweisung 3>>
}
else {
   <<Einzelne Anweisung 4>>
}
```

```
while ( ..... )
   <<Einzelne Anweisung 1>>

for ( .....; .....; ..... )
   <<Einzelne Anweisung 2>>

if ( ..... )
   <<Einzelne Anweisung 3>
else
   <<Einzelne Anweisung 4>>
```

In dieser Gegenüberstellung sehen Sie eine kleine mögliche Vereinfachung: Wenn in einem Rumpf einer Schleife oder im if-Teil oder im else-Teil nur eine einzelne Anweisung steht, dann dürfen die geschweiften Klammern auch jeweils weggelassen werden. Diese einzelne Anweisung kann auch wiederum eine Schleife oder eine Verzweigung sein.

Bei der Verzweigung gilt das für den if-Teil und den else-Teil unabhängig voneinander: Wenn der if-Teil nur eine Anweisung enthält, dann dürfen die geschweiften Klammern um diese Anweisung weggelassen werden, egal wie der else-Teil aussieht, und umgekehrt.

Anweisungen abarbeiten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if ( <<Bedingung 1>> )
   if ( <<Bedingung 2>> ) {
      <<Anweisung 1>>
   }
   else {
      <<Anweisung 2>>
   }
<<Anweisung 3>>
```

1. Falls Bedingung 1 nicht erfüllt ist, springe zu 6
2. Falls Bedingung 2 nicht erfüllt ist, springe zu 5
3. Führe Anweisung 1 aus
4. Springe zu 6
5. Führe Anweisung 2 aus
6. Führe Anweisung 3 aus

Wichtig ist bei zwei ineinander geschachtelten Verzweigungen, zu welchem if das else gehört, hier ein illustratives Beispiel.

Anweisungen abarbeiten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if ( <<Bedingung 1>> )
   if ( <<Bedingung 2>> ) {
      <<Anweisung 1>>
   }
   else {
      <<Anweisung 2>>
   }
<<Anweisung 3>>
```

1. Falls Bedingung 1 nicht erfüllt ist, springe zu 6
2. Falls Bedingung 2 nicht erfüllt ist, springe zu 5
3. Führe Anweisung 1 aus
4. Springe zu 6
5. Führe Anweisung 2 aus
6. Führe Anweisung 3 aus

Das else gehört immer zum letzten if, so wie hier durch die farbliche Unterlegung angezeigt. Sie können das auch auf der rechten Seite Schritt für Schritt nachvollziehen.

Anweisungen abarbeiten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if ( <<Bedingung 1>> )
   if ( <<Bedingung 2>> ) {
      <<Anweisung 1>>
   }
else {
   <<Anweisung 2>>
}
<<Anweisung 3>>
```

1. Falls Bedingung 1 nicht erfüllt ist, springe zu 6
2. Falls Bedingung 2 nicht erfüllt ist, springe zu 5
3. Führe Anweisung 1 aus
4. Springe zu 6
5. Führe Anweisung 2 aus
6. Führe Anweisung 3 aus

Das führt mitunter zu Fehlern, die man leicht übersieht: Wenn das else so wie hier eingerückt ist, dann sieht es so aus, als würde es zum ersten if gehören. Dem Compiler ist aber die Einrückung egal, der bezieht das else auf das letzte if, egal wie es eingerückt ist.

Anweisungen abarbeiten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if ( <<Bedingung 1>> ) {
   if ( <<Bedingung 2>> ) {
      <<Anweisung 1>>
   }
}
else {
   <<Anweisung 2>>
}
<<Anweisung 3>>
```

1. Falls Bedingung 1 nicht erfüllt ist, springe zu 5 !!!
2. Falls Bedingung 2 nicht erfüllt ist, springe zu 6 !!!
3. Führe Anweisung 1 aus
4. Springe zu 6
5. Führe Anweisung 2 aus
6. Führe Anweisung 3 aus

Um zu erzwingen, dass das else zum vorangehenden if gehört, kann man das zweite if doch in geschweifte Klammern setzen und somit vom else trennen. Sie können den Unterschied zum vorhergehenden Beispiel auch rechts nachvollziehen: Die Sprungziele sind jetzt vertauscht.

Anweisungen abarbeiten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
if ( <<Bedingung 1>> )
   if ( <<Bedingung 2>> )
      <<Anweisung 1>>
  else
     <<Anweisung 2>>
else
   if ( <<Bedingung 3>> )
     <<Anweisung 3>>
  else
     <<Anweisung 4>>
<<Anweisung 5>>
```

1. Falls Bedingung 1 nicht erfüllt ist, springe zu 7
2. Falls Bedingung 2 nicht erfüllt ist, springe zu 5
3. Führe Anweisung 1 aus
4. Springe zu 11
5. Führe Anweisung 2 aus
6. Springe zu 11
7. Falls Bedingung 3 nicht erfüllt ist, springe zu 10
8. Führe Anweisung 3 aus
9. Springe zu 11
10. Führe Anweisung 4 aus
11. Führe Anweisung 5 aus

Jetzt noch der Fall, dass mehrere if und else in der Verzweigung vorkommen. In diesem Beispiel gibt es vier Fälle: Wenn Bedingung 1 erfüllt ist, ist die Frage relevant, ob Bedingung 2 erfüllt ist, und wenn Bedingung 1 *nicht* erfüllt ist, ist statt dessen die Frage relevant, ob Bedingung 3 erfüllt ist. Mit dem bisher Gesagten sollten Sie in der Lage sein, diese vier Fälle auf der rechten Seite selbstständig durchzugehen, um so die linke Seite unmissverständlich zu verstehen.

Als weitere Übung können Sie sich auch überlegen, wie die rechte Seite zu reduzieren wäre, wenn links eines der beiden inneren if oder beide kein else haben. Und Sie können sich überlegen, was wäre, wenn Anweisung 1 bis 4 ebenfalls noch Verzweigungen oder Schleifen wären.

Demo-Beispiele ohne FopBot

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int sum = 0;
for ( int i = 1; i < 10; i++ )
   sum += i * i;
```

```
int value = 0;
int i = 1;
while ( value < 10000 ) {
   if ( i % 2 == 0 )
      value += i * i;
   else
      value –= i * i;
   i += 3;
}
```

```
int sum = 0;
for ( int i = 0; i < 100; i++ )
   if ( i * i % 50 < 20 ) {
      int max = 0;
      int j = i;
      while ( 357 % j < 347 ) {
         int tmp = i * j;
         if ( tmp > max )
            max = tmp;
         j += i;
      }
   }
```

Damit Ihre Vorstellung von Java nicht an FopBot kleben bleibt, schauen wir uns auf dieser Folie drei Demo-Beispiele *ohne* FopBot an, auch hier wieder ohne das ganze Brimborium mit public und class und main und so weiter drumherum, das bei Java ja eigentlich immer dabei sein muss.

***Demo-Beispiele* heißt, dass diese drei Beispiele nicht unbedingt etwas Sinnvolles berechnen, sondern den alleinigen Zweck haben, die Verwendung der verschiedenen Java-Konzepte zu demonstrieren. Wir haben noch nicht genug von Java gesehen, um Beispiele zu betrachten, die wirklich etwas Sinnvolles berechnen.**

Auch das ist eine grundlegende Kompetenz in der Informatik: Konstrukte und Beispiele für diese Konstrukte zu betrachten, ohne dass ein motivierender Anwendungshintergrund mitgeliefert wird.

Demo-Beispiele ohne FopBot

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int sum = 0;
for ( int i = 1; i < 10; i++ )
   sum += i * i;
```

```
int value = 0;
int i = 1;
while ( value < 10000 ) {
   if ( i % 2 == 0 )
      value += i * i;
   else
      value –= i * i;
   i += 3;
}
```

```
int sum = 0;
for ( int i = 0; i < 100; i++ )
   if ( i * i % 50 < 20 ) {
      int max = 0;
      int j = i;
      while ( 357 % j < 347 ) {
         int tmp = i * j;
         if ( tmp > max )
            max = tmp;
         j += i;
      }
   }
```

Das erste und einfachste Beispiel demonstriert nebenher das allgemeine Programmiermuster, wie man irgendwelche Werte aufsummiert: Man richtet eine Variable ein, in der schrittweise die Gesamtsumme aufgebaut wird, und initialisiert diese mit dem neutralen Element der Addition, also der 0 (will man Werte multiplizieren statt addieren, wählt man entsprechend das neutrale Element der Multiplikation, also die 1). In einer Schleife werden dann die Summanden draufaddiert, in jedem Durchlauf einer.

Demo-Beispiele ohne FopBot

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int sum = 0;
for ( int i = 1; i < 10; i++ )
   sum += i * i;
```

```
int value = 0;
int i = 1;
while ( value < 10000 ) {
   if ( i % 2 == 0 )
      value += i * i;
   else
      value –= i * i;
   i += 3;
}
```

```
int sum = 0;
for ( int i = 0; i < 100; i++ )
   if ( i * i % 50 < 20 ) {
      int max = 0;
      int j = i;
      while ( 357 % j < 347 ) {
         int tmp = i * j;
         if ( tmp > max )
            max = tmp;
         j += i;
      }
   }
```

Das zweite Beispiel demonstriert eine if-Verzweigung innerhalb einer Schleife.

Demo-Beispiele ohne FopBot

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int sum = 0;
for ( int i = 1; i < 10; i++ )
   sum += i * i;
```

```
int value = 0;
int i = 1;
while ( value < 10000 ) {
   if ( i % 2 == 0 )
      value += i * i;
   else
      value –= i * i;
   i += 3;
}
```

```
int sum = 0;
for ( int i = 0; i < 100; i++ )
   if ( i * i % 50 < 20 ) {
      int max = 0;
      int j = i;
      while ( 357 % j < 347 ) {
         int tmp = i * j;
         if ( tmp > max )
            max = tmp;
         j += i;
      }
   }
```

In jedem Durchlauf ändert i seine Parität gerade / ungerade. Dadurch kommt der Prozess in jedem zweiten Durchlauf durch die Schleife in den if-Teil, in den anderen Durchläufen in den else-Teil.

Demo-Beispiele ohne FopBot

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int sum = 0;
for ( int i = 1; i < 10; i++ )
   sum += i * i;
```

```
int value = 0;
int i = 1;
while ( value < 10000 ) {
   if ( i % 2 == 0 )
      value += i * i;
   else
      value –= i * i;
   i += 3;
}
```

```
int sum = 0;
for ( int i = 0; i < 100; i++ )
   if ( i * i % 50 < 20 ) {
      int max = 0;
      int j = i;
      while ( 357 % j < 347 ) {
         int tmp = i * j;
         if ( tmp > max )
            max = tmp;
         j += i;
      }
   }
```

Da es mühselig ist, zu berechnen, bei welchem i der Wert value die Zehntausendermarke knackt, bietet sich eine while-Schleife eher als eine for-Schleife an.

Demo-Beispiele ohne FopBot

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int sum = 0;
for ( int i = 1; i < 10; i++ )
   sum += i * i;
```

```
int value = 0;
int i = 1;
while ( value < 10000 ) {
   if ( i % 2 == 0 )
      value += i * i;
   else
      value –= i * i;
   i += 3;
}
```

```
int sum = 0;
for ( int i = 0; i < 100; i++ )
   if ( i * i % 50 < 20 ) {
      int max = 0;
      int j = i;
      while ( 357 % j < 347 ) {
         int tmp = i * j;
         if ( tmp > max )
            max = tmp;
         j += i;
      }
   }
```

Im dritten Beispiel sind eine Schleife, eine if-Verzweigung und wieder eine Schleife ineinander geschachtelt.

Demo-Beispiele ohne FopBot

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int sum = 0;
for ( int i = 1; i < 10; i++ )
   sum += i * i;
```

```
int value = 0;
int i = 1;
while ( value < 10000 ) {
   if ( i % 2 == 0 )
      value += i * i;
   else
      value –= i * i;
   i += 3;
}
```

```
int sum = 0;
for ( int i = 0; i < 100; i++ )
   if ( i * i % 50 < 20 ) {
      int max = 0;
      int j = i;
      while ( 357 % j < 347 ) {
         int tmp = i * j;
         if ( tmp > max )
            max = tmp;
         j += i;
      }
   }
```

Das ist das allgemeine Programmiermuster für die Berechnung des Maximums aus einer Reihe von Zahlen. Analog zur Berechnung der Summe aus Zahlen wird eine Variable eingerichtet und mit dem neutralen Element der Maximumsbildung initialisiert. Das ist eigentlich minus unendlich, aber da alle hier vorkommenden Zahlen nichtnegativ sind, kann auch die 0 – wie im übrigen auch jede beliebige negative Zahl – als neutrales Element dienen.

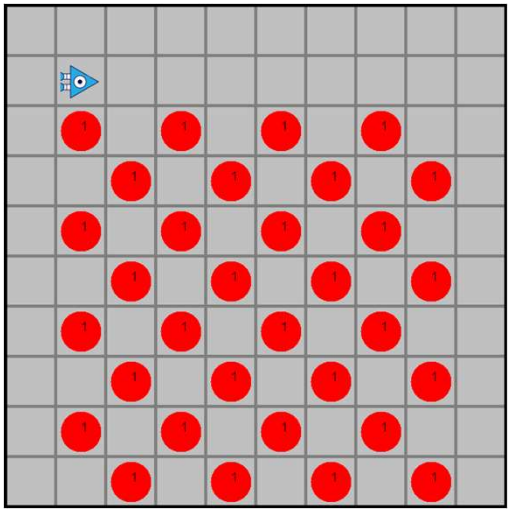
TECHNISCHE UNIVERSITÄT DARMSTADT

Rechteck nicht mehr vollständig, sondern schachbrettartig füllen

Zurück zu FopBot: Wir variieren jetzt das Beispiel, ein gefülltes Rechteck zu zeichnen.

Schachbrett zeichnen

TECHNISCHE UNIVERSITÄT DARMSTADT



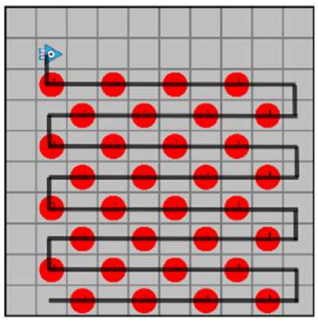
Und zwar *so*, dass dieses Bild herauskommt, also ein Schachbrett aus Münzen, wenn man so will.

Die Vorgehensweise

TECHNISCHE UNIVERSITÄT DARMSTADT

Wie wollen wir vorgehen?

- Bisher zeilenweise von links nach rechts
  - ➢ Der Robot musste umständlich positioniert werden
- Neue Idee: viermal immer abwechselnd von links nach rechts und von rechts nach links
  - ➢ Kein umständliches Positionieren mehr



- Noch zu klären: Wie dafür sorgen, dass genau die richtigen Felder mit Münzen belegt werden?
- Zielführende Beobachtung: Münzen gehören genau auf die Felder, bei denen die Summe aus Nummer der Zeile und Nummer der Spalte geradzahlig ist

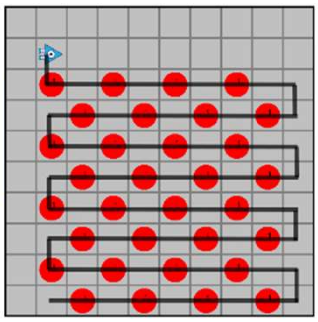
Wir machen uns vorher erst einmal wieder Gedanken über unsere Vorgehensweise. Dazu behalten wir oben rechts das Ziel, das wir erreichen wollen, im Auge.

Die Vorgehensweise

TECHNISCHE UNIVERSITÄT DARMSTADT

Wie wollen wir vorgehen?

- Bisher zeilenweise von links nach rechts
  - ➢ Der Robot musste umständlich positioniert werden
- Neue Idee: viermal immer abwechselnd von links nach rechts und von rechts nach links
  - ➢ Kein umständliches Positionieren mehr



- Noch zu klären: Wie dafür sorgen, dass genau die richtigen Felder mit Münzen belegt werden?
- Zielführende Beobachtung: Münzen gehören genau auf die Felder, bei denen die Summe aus Nummer der Zeile und Nummer der Spalte geradzahlig ist

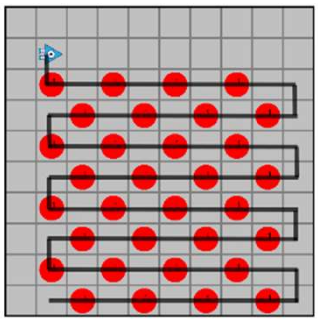
Unsere Vorgehensweise vom Zeichnen eines Rechtecks würde hier natürlich eins-zu-eins ebenfalls funktionieren. Aber vielleicht bekommen wir ja etwas Besseres hin?

Die Vorgehensweise

TECHNISCHE UNIVERSITÄT DARMSTADT

Wie wollen wir vorgehen?

- Bisher zeilenweise von links nach rechts
  - ➢ Der Robot musste umständlich positioniert werden
- Neue Idee: viermal immer abwechselnd von links nach rechts und von rechts nach links
  - ➢ Kein umständliches Positionieren mehr



- Noch zu klären: Wie dafür sorgen, dass genau die richtigen Felder mit Münzen belegt werden?
- Zielführende Beobachtung: Münzen gehören genau auf die Felder, bei denen die Summe aus Nummer der Zeile und Nummer der Spalte geradzahlig ist

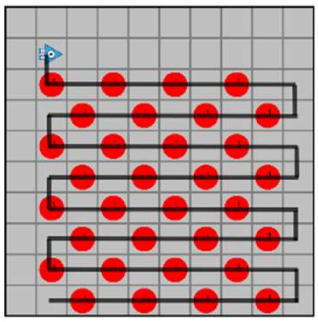
Eher ungefähr so wie hier farblich unterlegt würde man doch mit einem Stift durch die einzelnen Felder des Schachbretts gehen, ohne Leerlauf für den Rückweg von rechts nach links.

Die Vorgehensweise

TECHNISCHE UNIVERSITÄT DARMSTADT

Wie wollen wir vorgehen?

- Bisher zeilenweise von links nach rechts
  - ➢ Der Robot musste umständlich positioniert werden
- Neue Idee: viermal immer abwechselnd von links nach rechts und von rechts nach links
  - ➢ Kein umständliches Positionieren mehr



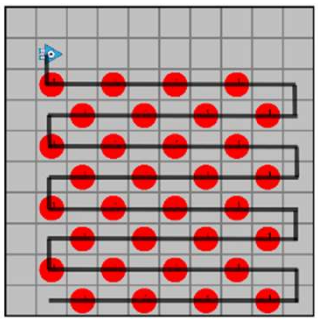
- Noch zu klären: Wie dafür sorgen, dass genau die richtigen Felder mit Münzen belegt werden?
- Zielführende Beobachtung: Münzen gehören genau auf die Felder, bei denen die Summe aus Nummer der Zeile und Nummer der Spalte geradzahlig ist

Beim Rechteck zeichnen war die Strategie zum Ablegen von Münzen einfach: Auf jedem Feld, das zum Rechteck gehört, wird jeweils eine Münze platziert. Aber nun soll nur auf jedem *zweiten* Feld jeweils eine Münze platziert werden, und das auch noch in jeder Zeile jeweils um eins gegenüber der vorhergehenden Zeile versetzt.

Die Vorgehensweise

TECHNISCHE UNIVERSITÄT DARMSTADT

Wie wollen wir vorgehen?



- Bisher zeilenweise von links nach rechts
  - ➢ Der Robot musste umständlich positioniert werden
- Neue Idee: viermal immer abwechselnd von links nach rechts und von rechts nach links
  - ➢ Kein umständliches Positionieren mehr
- Noch zu klären: Wie dafür sorgen, dass genau die richtigen Felder mit Münzen belegt werden?
- Zielführende Beobachtung: Münzen gehören genau auf die Felder, bei denen die Summe aus Nummer der Zeile und Nummer der Spalte geradzahlig ist

Sie glauben die farblich Beobachtung nicht? Prüfen Sie es nach!

Schachbrett zeichnen

TECHNISCHE UNIVERSITÄT DARMSTADT

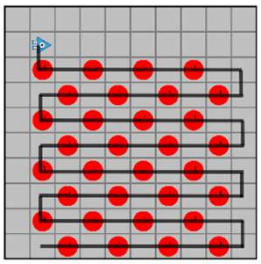
```
import fopbot.*;
import static Direction.*;

public class Chess implements Directions {
    public static void main ( String[] args ) {
        World.setVisible ( true );
        .....
        .....
        .....
    }
}
```

Die Datei heißt sinnigerweise Chess.java.

Schachbrett zeichnen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Robot chess = new Robot ( 1, 0, RIGHT, 32 );
for ( ... 4 ... ) {
    !! ausfuellen !!
}
```

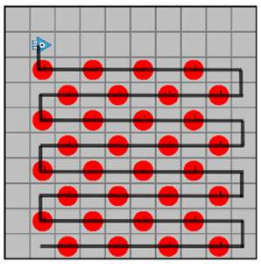


Insgesamt brauchen wir 32 Münzen.

Schachbrett zeichnen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Robot chess = new Robot ( 1, 0, RIGHT, 32 );
for ( ... 4 ... ) {
    !! ausfuellen !!
}
```

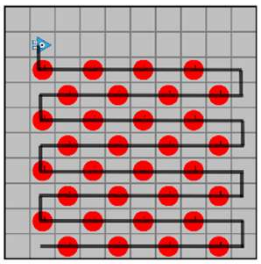


Wir werden insgesamt viermal hin- und herlaufen. Das heißt, in jedem Durchlauf durch die äußere Schleife erstellen wir zwei unmittelbar untereinander stehende Zeilen des Schachbretts.

Schachbrett zeichnen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Robot chess = new Robot ( 1, 0, RIGHT, 32 );
for ( ... 4 ... ) {
    !! ausfuellen !!
}
```

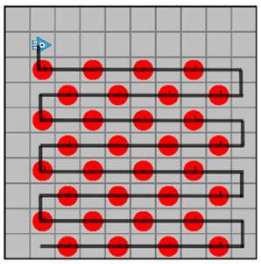


Das passiert also alles innerhalb dieser viermal durchlaufenen Schleife. Nach der Schleife ist das Programm zu Ende.

Schachbrett zeichnen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( ... 4 ... ) {
    for ( ... 8 ... ) {
        !! ausfuellen !!
    }
    !! ausfuellen !!
}
```

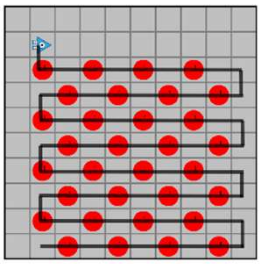


Hier ist nochmals die äußere for-Schleife, die auch auf der letzten Folie zu sehen war.

Schachbrett zeichnen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( ... 4 ... ) {
   for ( ... 8 ... ) {
      !! ausfuellen !!
      chess.move();
   }
   !! ausfuellen !!
}
```

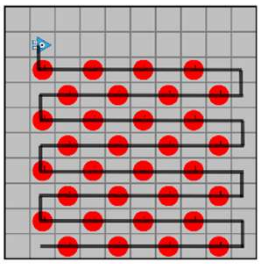


Um durch alle Spalten des Schachbretts zu gehen, müssen wir acht Schritte vorwärtsgehen. In dieser inneren Schleife wird eine Zeile des Schachbretts von links nach rechts, Spalte für Spalte durchlaufen.

Schachbrett zeichnen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( ... 4 ... ) {
   for ( ... 8 ... ) {
      !! ausfuellen !!
      chess.move();
   }
   !! ausfuellen !!
}
```



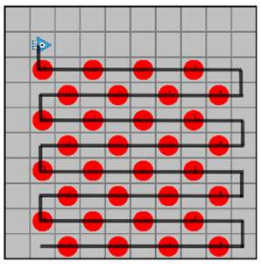
Die zugehörige zweite Zeile wird danach von rechts nach links durchlaufen. Zudem brauchen wir noch Anweisungen, um den Roboter nach jeder Zeile für einen Durchlauf eine Zeile höher in umgekehrter Richtung zu justieren.

Aber zunächst einmal bleiben wir noch beim Zeilendurchlauf von links nach rechts.

Schachbrett zeichnen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( ... 4 ... ) {
   for ( ... 8 ... ) {
      if ( !! ausfuellen !! )
         chess.putCoin();
      chess.move();
   }
   !! ausfuellen !!
}
```



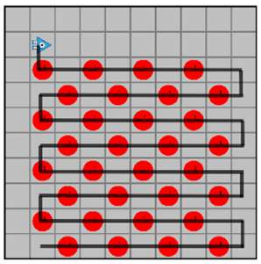
Hier kommt etwas Neues: Es soll ja nicht auf *jedem* Feld eine Münze platziert werden, sondern nur auf jeder *zweiten*.

Wir kennen schon ein Sprachkonstrukt, mit dem wir das Ausführen von Anweisungen von einer Bedingung abhängig machen können: die Verzweigung mit if.

Schachbrett zeichnen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( ... 4 ... ) {
    for ( ... 8 ... ) {
        if ( !! ausfuellen !! )
            chess.putCoin();
        chess.move();
    }
    !! ausfuellen !!
}
```



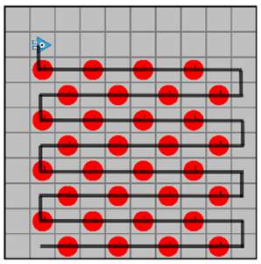
Um ein Schachbrett zu erzeugen, soll die farbig unterlegte Anweisung nur auf jedem *zweiten* Feld ausgeführt werden.

Wir haben schon gesehen: Wenn nur eine einzige Anweisung auszuführen ist, dann kann man die geschweiften Klammern weglassen. Hier lassen wir die geschweiften Klammern zur Abwechslung tatsächlich einmal weg.

Schachbrett zeichnen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( ... 4 ... ) {
    for ( ... 8 ... ) {
        if ( !! ausfuellen !! )
            chess.putCoin();
        chess.move();
    }
    !! ausfuellen !!
}
```

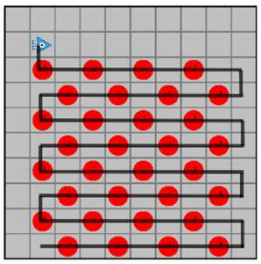


Die Bedingung steht in Klammern hinter dem Wort „if“. Nur wenn diese Bedingung erfüllt ist, wird eine Münze abgelegt.

Schachbrett zeichnen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( ... 4 ... ) {
    for ( ... 8 ... ) {
        if ( ( !! chess.getX() !! chess.getY() !! )
            chess.putCoin();
        chess.move();
    }
    !! ausfuellen !!
}
```

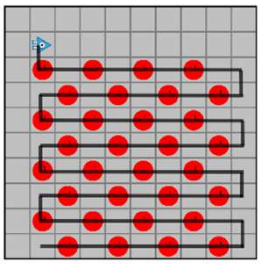


Unsere Idee basiert auf der aktuellen Spalte und der aktuellen Zeile des Roboters. Dafür bietet die Klasse Robot diese beiden Methoden an.

Schachbrett zeichnen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( ... 4 ... ) {
    for ( ... 8 ... ) {
        if ( ( !! chess.getX() !! chess.getY() !! )
            chess.putCoin();
        chess.move();
    }
    !! ausfuellen !!
}
```

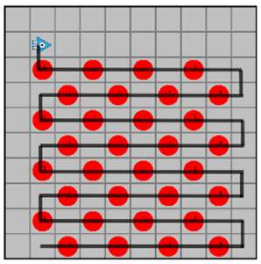


Die Rückgabewerte der beiden Methoden müssen dann noch miteinander verknüpft werden. Momentan wissen wir noch nicht, wie das aussehen soll. Deshalb packen wir eine Erinnerungsstütze an jede Stelle, wo vielleicht noch etwas fehlen *könnte*. Allerdings schreiben wir hier eine kürzere Erinnerungsstütze hin, nur zwei Ausrufezeichen, denn würden wir wie bisher „!! ausfuellen !!“ an jede dieser drei Stellen schreiben, wäre die Zeile wohl ein wenig lang und unübersichtlich.

Schachbrett zeichnen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( ... 4 ... ) {
    for ( ... 8 ... ) {
        if ( !! chess.getX() + chess.getY() !! )
            chess.putCoin();
        chess.move();
    }
    !! ausfuellen !!
}
```



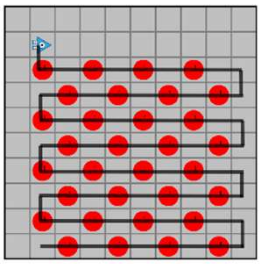
Wir hatten schon die Idee gehabt, die Parität der Summe aus Zeilen- und Spaltennummer als Kriterium herzunehmen, ob eine Münze auf einem Feld platziert wird oder nicht. Also setzen wir hier eine Addition ein.

Bleibt also nur noch die Frage, wie man die Parität testen kann, und dann wäre der Durchlauf durch eine Zeile von links nach rechts fertig.

Schachbrett zeichnen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( ... 4 ... ) {
    for ( ... 8 ... ) {
        if ( ( chess.getX() + chess.getY() ) % 2 == 0 )
            chess.putCoin();
        chess.move();
    }
    !! ausfuellen !!
}
```



Die Summe aus der momentanen Zeile und der momentanen Spalte des Roboters wird modulo 2 genommen und das Ergebnis mit 0 auf Gleichheit getestet.

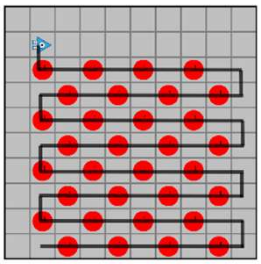
Der Rest einer Division durch 2 kann nur 0 oder 1 sein. Ist es 0, dann ist die Summe ohne Rest durch 2 teilbar, also eine gerade Zahl. Ist das Ergebnis der Module-Rechnung hingegen 1, dann ist die Summe eine ungerade Zahl.

Der entscheidende Punkt ist: In jedem Durchlauf durch die innere Schleife wird die Spalte um 1 größer, aber die Zeile bleibt dieselbe. Die Summe wird also in jedem Durchlauf um genau 1 größer und wechselt daher immer zwischen gerade und ungerade hin und her.

Schachbrett zeichnen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( ... 4 ... ) {
    for ( ... 8 ... ) {
        if ( ( chess.getX() + chess.getY() ) % 2 == 0 )
            chess.putCoin();
        chess.move();
    }
    !! ausfuellen !!
}
```

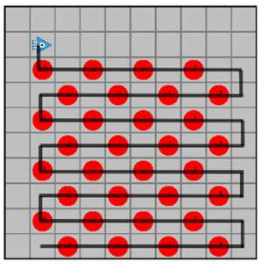


Hier vorne hatten wir uns ebenfalls eine Erinnerungsstütze gesetzt, stellen jetzt aber fest, dass hier nichts mehr einzufügen ist, die Schleife zum Durchlaufen einer Zeile von links nach rechts ist fertig.

Schachbrett zeichnen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( ... 4 ... ) {
    .........
    chess.turnLeft();
    chess.move();
    chess.turnLeft();
    .........
}
```



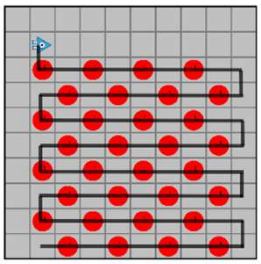
Nach einem Durchlauf durch das Schachbrett von links nach rechts steht der Roboter in der Spalte mit Index 9 und schaut nach rechts. Wir müssen ihn jetzt in die nächsthöhere Zeile bugsieren, und dort muss er in derselben Spalte stehen und nach links schauen, um dann die nächste Zeile von rechts nach links zu durchlaufen.

Nach allem, was wir bisher gesehen haben, können Sie sich leicht selbst davon überzeugen, dass diese drei Anweisungen zusammen genau das leisten.

Schachbrett zeichnen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( ... 4 ... ) {
    ..........
    for ( ... 8 ... ) {
        chess.move();
        if ( ( chess.getX() + chess.getY() )
            chess.putCoin();
    }
}
```



Der Durchgang durch die nächste Zeile von rechts nach links ist Zeichen für Zeichen ungefähr identisch zum Durchgang soeben von links nach rechts. Aber der Roboter steht zu Beginn nun in der Spalte rechts vom Schachbrett. Das heißt, in jedem Schritt muss nun zuerst der Roboter vorwärtbewegt werden, und erst danach wird die Münze gegebenenfalls platziert.

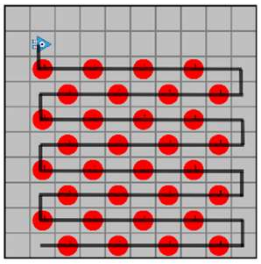
Machen Sie sich klar: Auf der nächsten Zeile ist die Zeilennummer um genau 1 größer. Das heißt, in den Spalten, bei denen die Summe in einer Zeile gerade ist, ist sie in der nächsten Zeile ungerade und umgekehrt.

Daher sind die Münz-Muster zweier aufeinanderfolgender Zeilen genau um 1 versetzt, was ja auch die Logik eines Schachbretts ist.

Schachbrett zeichnen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
.....
chess.turnLeft()
chess.turnLeft();
chess.turnLeft();
chess.move();
chess.turnLeft();
chess.turnLeft();
chess.turnLeft();
```



Am Ende eines Durchlaufs von rechts nach links steht der Roboter in der ersten Spalte und schaut nach links. Analog zu eben müssen wir den Roboter auf die nächste Zeile bugsieren, so dass er auf der ersten Spalte steht und nach rechts schaut.

Offensichtlich ist das völlig identisch zu eben, nur dass Linksdrehung durch Rechtsdrehung ersetzt werden muss.

Rechtsdrehung simulieren wir durch dreimal Linksdrehung.

TECHNISCHE UNIVERSITÄT DARMSTADT

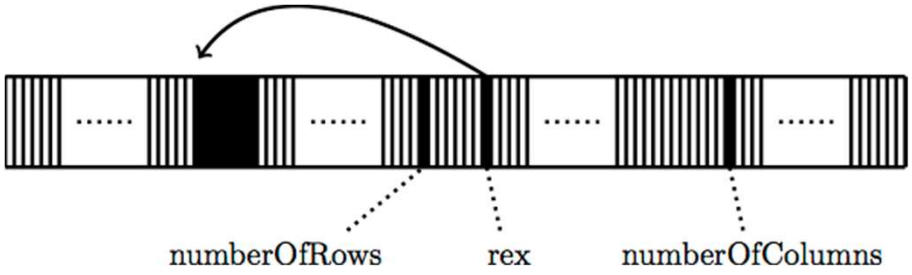
Begriffsunterscheidung:

Variable, Konstante, Referenz und Objekt

Bevor wir weitermachen, arbeiten wir kurz die verschiedenen Begriffe für Daten im Speicher auf und unterscheiden von jetzt an strikt zwischen den verschiedenen Arten von Daten im Speicher.

Begriffsklärung: Variable & Co

TECHNISCHE UNIVERSITÄT DARMSTADT

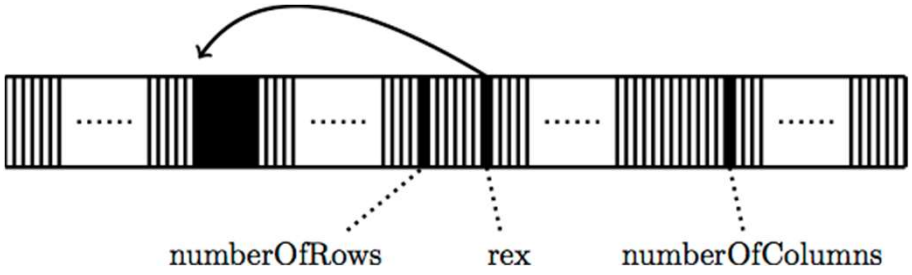


Variable: rex, numberOfRows und numberOfColumns
Referenz: rex
Objekt: worauf eine Referenz verweist

Für die Klärung von drei der vier Begriffe nehmen wir noch einmal das Bild von eben her, in dem wir den Unterschied zwischen Klassen und primitiven Datentypen betrachtet hatten.

Begriffsklärung: Variable & Co

TECHNISCHE UNIVERSITÄT DARMSTADT

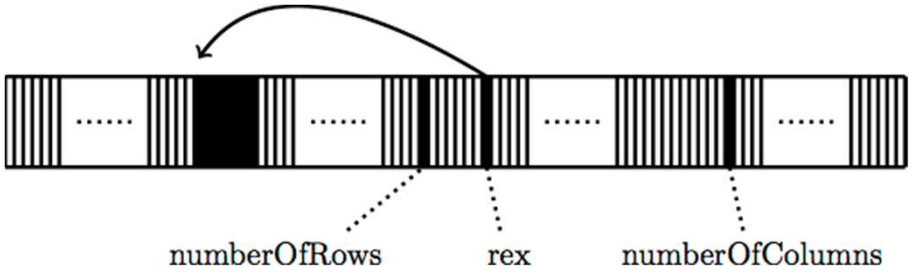


Variable: rex, numberOfRows und numberOfColumns
Referenz: rex
Objekt: worauf eine Referenz verweist

Die Daten, denen wir bisher selbst Namen gegeben hatten, sind Variablen. Der Name leitet sich aus dem Unterschied zu Konstanten her. Konstanten und den Unterschied zu Variablen sehen wir gleich.

Begriffsklärung: Variable & Co

TECHNISCHE UNIVERSITÄT DARMSTADT



Variable: rex, numberOfRows und numberOfColumns

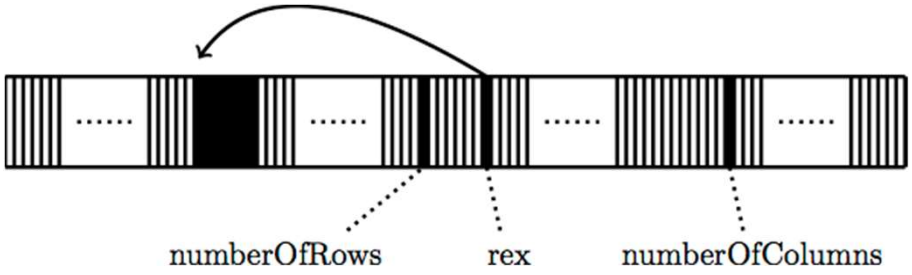
Referenz: rex

Objekt: worauf eine Referenz verweist

Eine Variable oder Konstante von einem Klassentyp nennen wir Referenz, weil sie ein Objekt *referenziert*.

Begriffsklärung: Variable & Co

TECHNISCHE UNIVERSITÄT DARMSTADT



Variable: rex, numberOfRows und numberOfColumns
Referenz: rex
Objekt: worauf eine Referenz verweist

Und nur das, was mit Operator new reserviert wird, nennen wir ein *Objekt*. Variable und Konstanten sind also *keine* Objekte, und Objekte sind *keine* Variablen oder Konstanten.

Variable und Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int m = 1;
m = 2;

final int N = 3;
N = 4;
```

Jetzt kommen wir noch zum Begriff Konstante und zum Unterschied zwischen Variable und Konstante. So wie in den ersten beiden Zeilen kennen wir es schon: Eine Variable namens m wird eingerichtet und initialisiert, und später wird ihr Wert noch einmal überschrieben.

Variable und Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int m = 1;
m = 2;

final int N = 3;
N = 4;
```

Dieses Schlüsselwort ist neu: Es besagt hier, dass N keine Variable, sondern eine Konstante ist.

Variable und Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int m = 1;
m = 2;

final int N = 3;
N = 4;
```

Der Unterschied ist, dass der Wert von N später nicht mehr geändert werden darf. Wenn Sie es wie hier versuchen, gibt der Compiler eine Fehlermeldung aus und übersetzt den Quelltext nicht.

Die Begriffe Variable und Konstante sowie das Schlüsselwort final sind also recht gut gewählt.

Variable und Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int m = 1;
m = 2;

final int N = 3;
N = 4;
```

***Erinnerung*: In Kapitel 01a, Abschnitt zu Identifiern, hatten wir schon die Konvention gesehen, dass die Namen von Konstanten aus Großbuchstaben gebildet werden.**

Variable und Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
final int CENTS_PER_EURO = 100;

.........

CENTS_PER_EURO = 200;
```

Konstanten sind dafür da, bestimmte Werte vor versehentlicher Änderung zu schützen, nämlich solche Werte, bei denen eine Änderung garantiert ein Fehler wäre. Dazu zählen etwa Naturkonstanten wie die Kreiszahl und die Eulersche Zahl oder auch definitorische Konstanten wie diese.

Variable und Konstanten

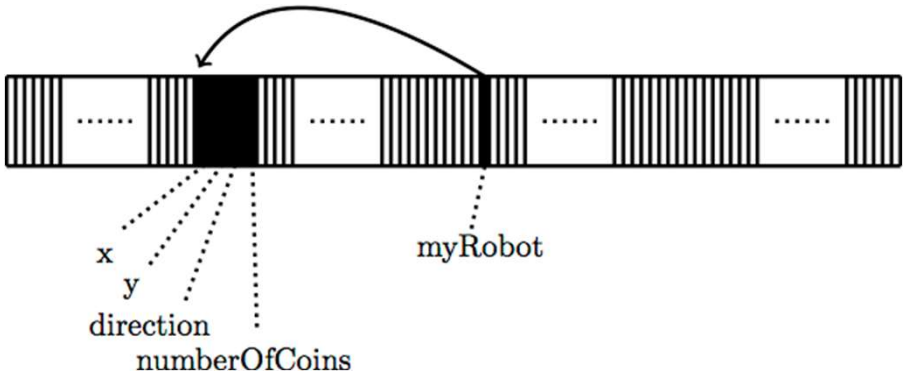
TECHNISCHE UNIVERSITÄT DARMSTADT

```
final int CENTS_PER_EURO = 100;

.........

CENTS_PER_EURO = 200;
```

Die Konvention, bei Konstanten alle Buchstaben groß zu schreiben, erzwingt, dass auch die Konvention zur Kennzeichnung von Wortanfängen mitten im Identifier anders sein muss, denn wo alle Buchstaben groß sind, können Großbuchstaben nichts kennzeichnen. Die Konvention bei Konstanten ist: Einzelne Wörter im Namen werden durch Underscore getrennt.

Variable und Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

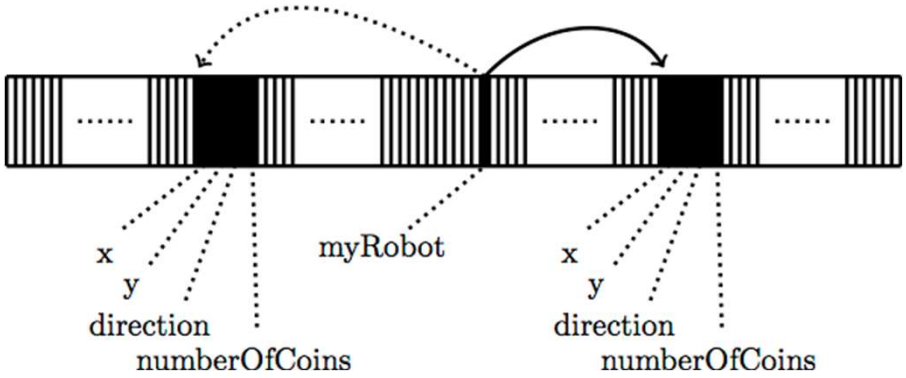


Robot myRobot = new Robot (4, 2, UP, 0);

Auch von Klassen hatten wir bisher nur Variable gesehen, noch keine Konstanten. Jetzt schauen wir uns auch Konstanten bei Klassen an. Wir gehen von der altbekannten Situation aus, die wir schon mehrfach gesehen hatten.

Variable und Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

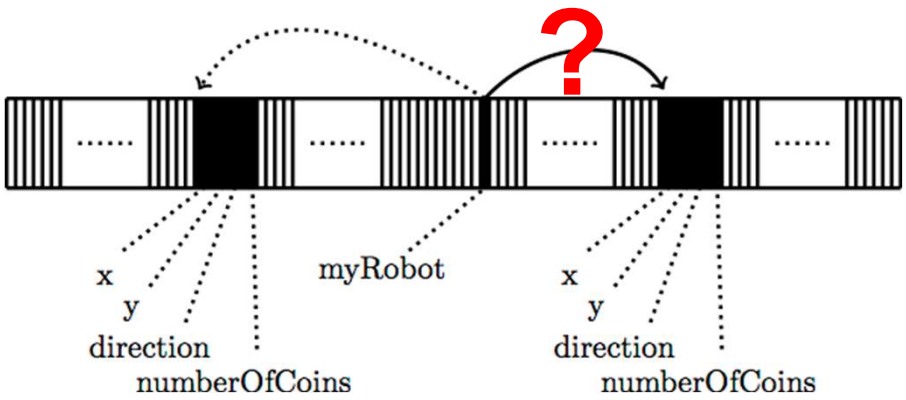



```
Robot myRobot = new Robot ( 4, 2, UP, 0 );
myRobot = new Robot ( 3, 5, DOWN, 3 );
```

Wir fügen eine zweite Anweisung ein, die die Referenz sozusagen „verbiegt“. Das linke Objekt sei das durch das erste new erzeugte, das rechte Objekt sei das durch das zweite new erzeugte. Der gestrichelte Pfeil im Bild deutet also den Inhalt von myRobot nach der *ersten* Anweisung unten auf der Folie an, der durchgezogene Pfeil hingegen den Inhalt von myRobot nach der *zweiten* Anweisung.

Variable und Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

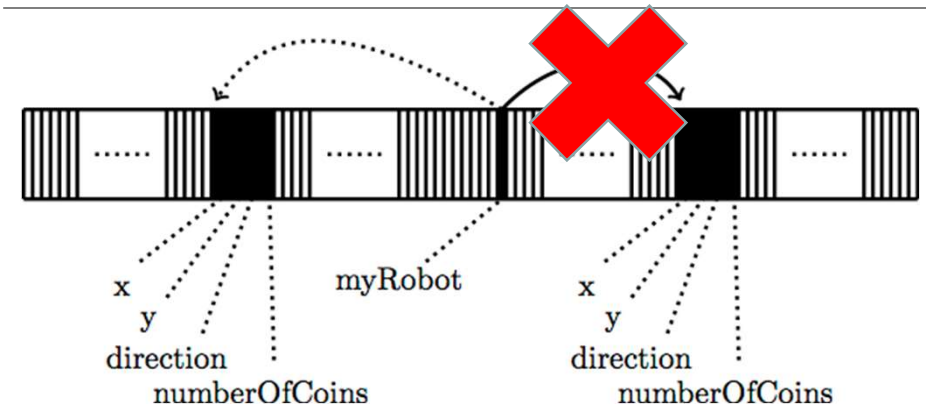



```
final Robot myRobot = new Robot ( 4, 2, UP, 0 );
myRobot = new Robot ( 3, 5, DOWN, 3 );
```

Jetzt setzen wir noch das Schlüsselwort final an dieselbe Stelle wie vorher bei dem primitiven Datentyp int und fragen uns, was das nun bei Referenztypen bedeutet.

Variable und Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

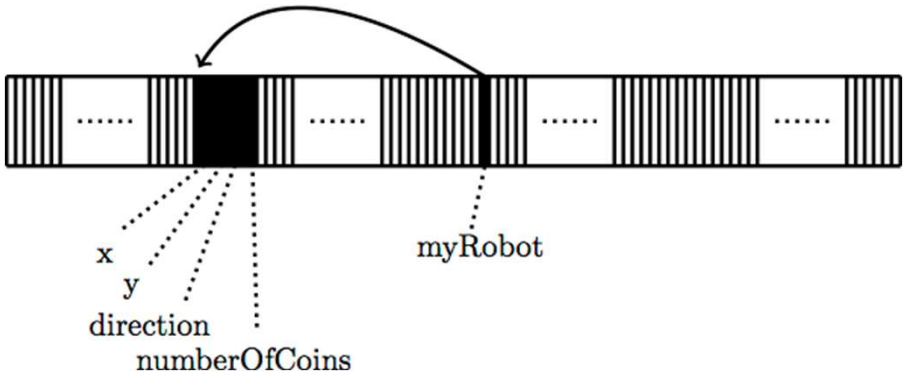



```
final Robot myRobot = new Robot ( 4, 2, UP, 0 );
myRobot = new Robot ( 3, 5, DOWN, 3 );
```

Die einzige inhaltliche Änderung ist also, dass myRobot nun eine Konstante ist, keine Variable mehr. Daher geht die zweite Anweisung nicht mehr durch den Compiler.

Variable und Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT




```
final Robot myRobot = new Robot ( 4, 2, UP, 0 );
myRobot.move();
```

Was hingegen weiterhin möglich ist, ist eine Änderung der Werte im *Objekt*. Durch Methode move wird ja entweder das Attribut x oder das Attribut y geändert, je nachdem, ob der Roboter gerade in eine horizontale oder in eine vertikale Richtung schaut.

Also zusammenfassend: Die Referenz selbst darf dank final nicht geändert werden, das referenzierte Objekt sehr wohl.

TECHNISCHE UNIVERSITÄT DARMSTADT

Werten immer Namen geben!

Nachdem wir Variable und Konstanten gesehen haben, sind wir soweit, dieses wichtige Grundprinzip der professionellen Softwareentwicklung anzusprechen. Der Titel des Abschnitts spricht eigentlich für sich, wir sehen uns das aber trotzdem an einem Beispiel an.

Variable und Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
final int  NUMBER_OF_SEASONS  =  4;

.........

final int  NUMBER_OF_MUSKETEERS  =  4;    // ???
```

Grundsätzlich sollte jede Zahl außer vielleicht 0, 1 und -1 zu einer Konstanten gemacht werden. Den Grund sehen Sie in diesem fiktiven Beispiel. Stellen Sie sich vor, Sie haben ein Programm mit mehreren Millionen Zeilen, in dem die Zahl 4 ein paar tausend mal vorkommt, in der einen Hälfte der Fälle in der Bedeutung 4 Jahreszeiten, in der anderen Hälfte der Fälle in der Bedeutung 4 Musketiere.

Variable und Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
final int  NUMBER_OF_SEASONS  =  4;

.........

final int  NUMBER_OF_MUSKETEERS  =  5;    // !!!
```

Irgendwann im Laufe der Jahre, in denen das Programm genutzt wird, kommt vielleicht ein fünfter Musketier hinzu, das heißt, die Anzahl Musketiere ist jetzt nicht mehr 4, sondern 5, die Anzahl der Jahreszeiten bleibt natürlich bei 4. Wenn Sie konsequent überall im Programm die Konstantennamen NUMBER_OF_SEASONS und NUMBER_OF_MUSKETEERS verwendet haben, dann ändern Sie die Anzahl der Musketiere einfach in dieser einen Zeile ab und fertig.

Falls Sie hingegen ein paar tausend mal die Zahl 4 direkt hingeschrieben haben, dann müssen Sie sich eben jedes Vorkommen der Zahl 4 genau anschauen und entscheiden, ob das jetzt 4 Jahreszeiten oder 4 Musketiere waren. Im ersten Fall bleibt die Zahl 4 unangetastet, im zweiten Fall wird sie um 1 erhöht. Wenn Sie auch nur bei einem einzigen der vielen tausend Vorkommen der Zahl 4 einen Fehler machen, dann ist die Wahrscheinlichkeit groß, dass dieser Fehler extrem schwer und aufwendig zu finden sein wird – vielleicht erst nach Auslieferung an den Kunden auftritt.

TECHNISCHE UNIVERSITÄT DARMSTADT

Allgemein:

Primitive Datentypen

Oracle Java Tutorials: Primitive Data Types

Oracle Java Spezifikation: 4.2 Primitive Types and Values

Wir haben bisher nur mit dem primitiven Datentyp int gearbeitet. Jetzt sehen wir uns kurz an, was es sonst noch so an primitiven Datentypen in Java gibt.

Wichtig ist zu beachten, dass alles hier Gesagte erst einmal nur für Java so gilt. Es gibt Datentypen gleichen Namens auch in anderen Programmiersprachen, aber in den Details gibt es oftmals erhebliche Unterschiede.

Übersicht

TECHNISCHE UNIVERSITÄT DARMSTADT

Ganze Zahlen:
- byte
- short
- int
- long

Gebrochene Zahlen:
- float
- double

Logik:
- boolean

Zeichen:
- char

Das sind die primitiven Datentypen in Java im Überblick.

Übersicht

TECHNISCHE UNIVERSITÄT DARMSTADT

Ganze Zahlen:
- byte
- short
- int
- long

Gebrochene Zahlen:
- float
- double

Logik:
- boolean

Zeichen:
- char

Zum ersten sind da vier verschiedene Typen für ganze Zahlen. Diese Typen heißen byte, short, int und long. Wie wir gleich sehen werden, unterscheiden sie sich eigentlich nur darin, mit wie vielen Bits eine Zahl gespeichert wird, und welche Zahlenbereiche durch die einzelnen Datentypen somit abgedeckt werden.

Übersicht

TECHNISCHE UNIVERSITÄT DARMSTADT

Ganze Zahlen:
- byte
- short
- int
- long

Gebrochene Zahlen:
- float
- double

Logik:
- boolean

Zeichen:
- char

Dann gibt es zwei verschiedene Typen für reelle Zahlen, float und double. Wir sagen gebrochene Zahlen, denn beliebige reelle Zahlen lassen sich natürlich nicht im Computer darstellen. Auch zwischen diesen beiden Datentypen ist der wesentliche Unterschied die Anzahl Bits beziehungsweise daraus resultierend der darstellbare Bereich.

Übersicht

TECHNISCHE UNIVERSITÄT DARMSTADT

Ganze Zahlen:
- byte
- short
- int
- long

Gebrochene Zahlen:
- float
- double

Logik:
- boolean

Zeichen:
- char

Ein Datentyp namens boolean für die Werte true und false und für logische Operationen.

Dieser Datentyp ist nach George Boole benannt, einem englischen Mathematiker, Logiker und Philosophen im neunzehnten Jahrhundert.

Übersicht

TECHNISCHE UNIVERSITÄT DARMSTADT

Ganze Zahlen:
- byte
- short
- int
- long

Gebrochene Zahlen:
- float
- double

Logik:
- boolean

Zeichen:
- char

Schießlich noch ein Datentyp für Schriftzeichen, genannt char, Abkürzung für englisch character, was unter anderem mit Schriftzeichen übersetzt werden kann.

Ganze Zahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

Typ	Bits	Bereich
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

Zuerst zu den vier ganzzahligen Typen: Wie gesagt, unterscheiden sich die einzelnen ganzzahligen Datentypen in der Größe. Ein Datentyp mit N Bits kann 2 hoch N viele unterschiedliche Zahlen darstellen.

Ganze Zahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

Typ	Bits	Bereich
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

Das sind also 2 hoch 8 gleich 256 unterschiedliche Zahlen beim Datentyp byte. Da natürlich die 0 dargestellt werden muss, sind insgesamt 255 unterschiedliche positive und negative Zahlen möglich. Die Anzahl darstellbarer positiver und die Anzahl darstellbarer negativer Zahlen kann daher nicht gleich sein.

***Vorgriff*: Warum diese beiden Anzahlen genau so sind, werden wir später in Kapitel 11 betrachten, wo wir genauer in die Kodierung der Zahlentypen hineinschauen werden.**

Ganze Zahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

Typ	Bits	Bereich
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

Der nächste Datentyp hat 16 Bit: 2 hoch 16 sind etwas mehr als fünfundsechzigtausend.

Wir verwenden auf diesen Folien die angelsächsische Notation von Zahlen, das heißt, die Bedeutung von Punkt und Komma sind genau andersherum wie in der deutschen Notation: Dreierkolonnen werden nicht durch Punkte, sondern durch Kommas separiert, und der ganzzahlige Anteil einer Zahl wird von dem, was wir im Deutschen „Nachkommastellen“ nennen, durch einen Punkt getrennt, den Dezimalpunkt.

Wir werden gleich bei float und double sehen, dass Java wie eigentlich so ziemlich jede Programmiersprache den Dezimalpunkt verwendet.

Ganze Zahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

Typ	Bits	Bereich
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

Der Datentyp int ist der Standarddatentyp für ganze Zahlen. Er ist eigentlich erstaunlich klein: Die Schulden einer deutschen Großstadt gerechnet in Euro passen nicht unbedingt in den Datentyp int. Man muss sich also schon zweimal überlegen, ob int für die Zwecke des eigenen Java-Programms ausreichend groß ist.

Ganze Zahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

Typ	Bits	Bereich
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

Der Datentyp long dürfte allerdings für so ziemlich alle praktischen Zwecke ausreichend groß sein.

Ganzzahlige Literale

TECHNISCHE UNIVERSITÄT DARMSTADT

123

+123

−123

Dies sind drei *ganzzahlige* Literale. Literale sind wörtlich hingeschriebene Werte eines Datentyps. Das englische Adjektiv „literal“ heißt unter anderem „wörtlich“.

Das Vorzeichen plus ändert zwar nichts am Wert, kann aber zuweilen die Lesbarkeit erhöhen, zum Beispiel wenn man im Finanzbereich konsequent Haben mit plus und Soll mit minus kennzeichnet.

Der Datentyp solcher Literale ist int.

Ganzzahlige Literale

TECHNISCHE UNIVERSITÄT DARMSTADT

123l	123L
+123l	+123L
−123l	−123L

Wie schon eingangs angedeutet, umfasst der Datentyp long mehr Speicherplatz als int und kann daher einen deutlich größeren Bereich von ganzen Zahlen als int darstellen.

Will man ein Literal vom Typ long, muss man ein kleines oder großes L ans Ende setzen. Man sollte immer ein großes L setzen, da das kleine l leicht mit der Ziffer 1 verwechselt werden kann,

Literale vom Typ long sind sinnvoll, wenn man einen Wert als Literal hinschreiben möchte, der nicht in den Datentyp int hineinpasst.

Ganzzahlige Literale

TECHNISCHE UNIVERSITÄT DARMSTADT

```
byte  b = 123;

short s = 123;
```

Die beiden anderen ganzzahligen primitiven Datentypen, byte und short, sind kleiner als int und nur für spezielle Zwecke gedacht. Diese beiden Datentypen haben keine eigenen Literale, lassen sich aber durch Literale vom Datentyp int initialisieren, sofern diese Literale klein genug sind, dass sie in den jeweiligen Datentyp passen. Bei der Zahl 123 ist das für beide Datentypen der Fall (der kleinere von beiden, Byte, kann ganze Zahlen bis 127 darstellen).

Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

Maximaler positiver/negativer int-Wert:

- **Integer.MAX_VALUE**
- **Integer.MIN_VALUE**

➔Analog byte (Byte), short (Short), long (Long)

Die oberen und unteren Grenzen der einzelnen Zahlentypen müssen wir uns zum Glück nicht merken, sie sind in symbolischen Konstanten schon vordefiniert. Oben sehen Sie die Namen der beiden Konstanten für den größtmöglichen positiven Wert und den betragsmäßig größtmöglichen negativen Wert im Datentyp int. Unten sehen Sie, welches Wort anstelle von Integer stehen muss, um die entsprechenden Konstanten für die anderen ganzzahligen Datentypen zu bekommen. Das Wort hinter dem Punkt ist jeweils dasselbe, MAX_VALUE beziehungsweise MIN_VALUE.

***Vorgriff*: die beiden Konstanten MAX_VALUE und MIN_VALUE gehören zu einer Klasse namens Integer, daher steht Integer davor. Systematisch sehen wir uns das in Kapitel 03a an, Abschnitt zu Klassenkonstanten.**

Ganze Zahlen: Überlauf

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int n = Integer.MAX_VALUE / 2 * 3;    //  n == ???
```

Was passiert in der beispielhaften Anweisung oben:

- **Kein Prozessabbruch, keine Fehlermeldung!**
- **Die arithmetischen Operationen werden blindlings durchgeführt**
  - **Die unteren 32 Bits des Ergebnisses werden in n gespeichert**
  - **Hochgefährlich → unbedingt vermeiden!**

Einen wichtigen Punkt hatten wir bisher noch nicht angesprochen: Was passiert eigentlich, wenn das Ergebnis einer arithmetischen Operation nicht mehr in den Datentyp hineinpasst? Das Beispiel oben auf der Folie ist so gewählt, dass der Rückgabetyp des Ausdrucks int ist, der berechnete Wert aber nicht in den Datentyp int hineinpasst.

Ganze Zahlen: Überlauf

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int n = Integer.MAX_VALUE / 2 * 3;    //  n == ???
```

Was passiert in der beispielhaften Anweisung oben:

- Kein Prozessabbruch, keine Fehlermeldung!
- Die arithmetischen Operationen werden blindlings durchgeführt
  - Die unteren 32 Bits des Ergebnisses werden in n gespeichert
  - Hochgefährlich → unbedingt vermeiden!

Die erste Antwort ist, dass die Ausführung des Programms in einem solchen Fall nicht abgebrochen wird, sondern das Programm läuft einfach weiter und verarbeitet auch den Wert des Ausdrucks einfach weiter.

Daraus resultiert die Anschlussfrage: Was ist der Wert dieses Ausdrucks, also der Wert von n nach der Zuweisung?

Ganze Zahlen: Überlauf

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int n = Integer.MAX_VALUE / 2 * 3;    //  n == ???
```

Was passiert in der beispielhaften Anweisung oben:

- Kein Prozessabbruch, keine Fehlermeldung!
- Die arithmetischen Operationen werden blindlings durchgeführt
  - ➢ Die unteren 32 Bits des Ergebnisses werden in n gespeichert
  - ➢ Hochgefährlich → unbedingt vermeiden!

Wir brauchen hier nicht ins Detail gehen, denn egal wie das Ergebnis eines arithmetischen Überlaufs genau aussieht – es kann nach menschlichem Ermessen praktisch nie das eigentlich intendierte Ergebnis sein. Daher sollten Überläufe vermieden werden, am Besten durch Wahl eines größeren Datentyps.

***Vorgriff*: Java bietet eine Klasse namens BigInteger für das Rechnen mit beliebig großen ganzzahligen Werten. Die Rechnungen sind damit aber viel langsamer als mit primitiven Datentypen, und Quelltexte mit BigInteger sind auch komplizierter zu schreiben und zu lesen.**

Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

	float	double
Bits	32	64
Max. Wert	$\pm$ 3.402... $\cdot$ 10^{38}	$\pm$ 1.797... $\cdot$ 10^{308}
Genauigkeit	1 zu 8.388.608	ca. 1 zu 4,5 Billiarden

Wir kommen nun zu den beiden gebrochenzahligen Typen: float und double. Ursprünglich war float als Standardtyp gedacht. Da aber Speicherplatz im Laufe der Zeit immer billiger geworden ist, hat der doppelt so speicherintensive Datentyp double diese Rolle übernommen.

***Vorgriff*: Auch zu diesen beiden Datentypen werden wir uns später, in Kapitel 11, ansehen, wie sie tatsächlich realisiert sind.**

Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

	float	double
Bits	32	64
Max. Wert	$\pm$ 3.402... $\cdot 10^{38}$	$\pm$ 1.797... $\cdot 10^{308}$
Genauigkeit	1 zu 8.388.608	ca. 1 zu 4,5 Billiarden

Wie bei ganzzahligen Datentypen ist der wesentliche Unterschied die Anzahl Bits. Der Name double steht übrigens für double precision, also doppelte Präzision, was nicht so ganz genau den Sachverhalt wiedergibt, wie die Zahlen unten zeigen.

Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

	float	double
Bits	32	64
Max. Wert	$\pm$ 3.402... $\cdot 10^{38}$	$\pm$ 1.797... $\cdot 10^{308}$
Genauigkeit	1 zu 8.388.608	ca. 1 zu 4,5 Billiarden

Der größte darstellbare Wert ist schon bei float jenseits der meisten praktischen Erfordernisse, bei double bleibt wohl kein Wunsch mehr übrig.

Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

	float	double
Bits	32	64
Max. Wert	$\pm$ 3.402... $\cdot 10^{38}$	$\pm$ 1.797... $\cdot 10^{308}$
Genauigkeit	1 zu 8.388.608	ca. 1 zu 4,5 Billiarden

Der Datentyp float kann zwei Zahlen, die um circa 1 differieren, nur unterscheiden, wenn die beiden Zahlen nicht größer als circa 8 Komma 4 Millionen sind.

Oder: Zwei Zahlen, die ungefähr gleich 1 sind, lassen sich nur unterscheiden, wenn sie höchstens um circa 8 Komma 4 Millionstel differieren.

Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

	float	double
Bits	32	64
Max. Wert	$\pm$ 3.402... $\cdot 10^{38}$	$\pm$ 1.797... $\cdot 10^{308}$
Genauigkeit	1 zu 8.388.608	ca. 1 zu 4,5 Billiarden

Bei double ist die Genauigkeit mehr als 500 Millionen mal höher.

Gebrochenzahlige Literale

TECHNISCHE UNIVERSITÄT DARMSTADT

12.34	.1234	1.2E34
+12.34	+.1234	+1.2E−34
−12.34	−.1234	−1.2E+34

Jetzt zu Literalen, die Zahlen darstellen, die nicht ganzzahlig sind. Diese sind vom Datentyp double, es sei denn, man setzt ein kleines oder großes F ans Ende analog zu kleinem oder großem L bei long. Mit einem kleinen oder großen F sind gebrochenzahlige Literale vom Typ float.

Gebrochenzahlige Literale

TECHNISCHE UNIVERSITÄT DARMSTADT

12.34	.1234	1.2E34
+12.34	+.1234	+1.2E−34
−12.34	−.1234	−1.2E+34

Wie allgemein üblich in der angelsächsischen Welt, gibt es auch in Java kein Komma, sondern einen Dezimal*punkt* zwischen dem ganzzahligen und dem nichtganzzahligen Anteil.

Gebrochenzahlige Literale

TECHNISCHE UNIVERSITÄT DARMSTADT

12.34	.1234	1.2E34
+12.34	+.1234	+1.2E−34
−12.34	−.1234	−1.2E+34

Geht natürlich auch wieder mit einem positiven beziehungsweise negativen Vorzeichen.

Gebrochenzahlige Literale

TECHNISCHE UNIVERSITÄT DARMSTADT

12.34	.1234	1.2E34
+12.34	+.1234	+1.2E−34
−12.34	−.1234	−1.2E+34

Wenn der ganzzahlige Anteil vor dem Dezimalpunkt 0 ist, darf diese 0 auch fehlen – hätte aber selbstverständlich auch hingeschrieben werden dürfen.

Gebrochenzahlige Literale

TECHNISCHE UNIVERSITÄT DARMSTADT

12.34	.1234	1.2E34
+12.34	+.1234	+1.2E−34
−12.34	−.1234	−1.2E+34

Auch hier natürlich wieder Vorzeichen möglich.

Gebrochenzahlige Literale

TECHNISCHE UNIVERSITÄT DARMSTADT

12.34	.1234	1.2E34
+12.34	+.1234	+1.2E−34
−12.34	−.1234	−1.2E+34

Alternativ gibt es die wissenschaftliche Notation. Damit bezeichnet man generell Zahlenangaben mit Exponenten. In Java sieht diese Notation so aus wie hier gezeigt. Der Wert dieses Literals ist 1 Komma 2 mal 10 hoch 34.

Gebrochenzahlige Literale

TECHNISCHE UNIVERSITÄT DARMSTADT

12.34	.1234	1.2E34
+12.34	+.1234	+1.2E−34
−12.34	−.1234	−1.2E+34

Und auch hier wieder positives oder negatives Vorzeichen möglich, bei jedem der beiden Bestandteile.

Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

Probleme mit Ungenauigkeiten (Beispiele):

- **Umkehrrechnungen liefern nicht genau den Ausgangswert**
- **Bei Addition extrem unterschiedlich großer Zahlen geht die kleinere unter**
- **Subtraktion fast gleich großer Zahlen kann mglw. 0 oder nur inkorrekte Bits ergeben**
- **Test auf Gleichheit muss ersetzt werden durch Test auf „ausreichend nahe beieinander“**

Die Genauigkeit insbesondere von double ist zwar beeindruckend hoch, aber jede noch so kleine Ungenauigkeit macht potentiell Probleme. In numerischen Berechnungen werden unzählige Male dieselben Berechnungen durchgeführt, wobei das Ergebnis einer Berechnung dann wieder Eingabe der nächsten Berechnung ist. Das hat häufig zur Folge, dass der numerische Fehler wächst und wächst und wächst.

Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

Probleme mit Ungenauigkeiten (Beispiele):

- **Umkehrrechnungen liefern nicht genau den Ausgangswert**
- **Bei Addition extrem unterschiedlich großer Zahlen geht die kleinere unter**
- **Subtraktion fast gleich großer Zahlen kann mglw. 0 oder nur inkorrekte Bits ergeben**
- **Test auf Gleichheit muss ersetzt werden durch Test auf „ausreichend nahe beieinander“**

Das klassische Beispiel für den farbig unterlegten Pnukt ist das Wurzelziehen. Wenn man beispielsweise die Quadratwurzel von 2 berechnet und das Ergebnis quadriert, sollte eigentlich wieder 2 herauskommen. Es kommt aber ein Wert heraus, der ein ganz klein wenig von der 2 abweicht. Wenn man davon wieder die Quadratwurzel zieht und das Ergebnis wieder quadriert, ist die Abweichung von 2 etwas größer geworden. Je öfters man das wiederholt, um so mehr weichen die Ergebnisse von 2 ab.

Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

Probleme mit Ungenauigkeiten (Beispiele):

- **Umkehrrechnungen liefern nicht genau den Ausgangswert**
- **Bei Addition extrem unterschiedlich großer Zahlen geht die kleinere unter**
- **Subtraktion fast gleich großer Zahlen kann mglw. 0 oder nur inkorrekte Bits ergeben**
- **Test auf Gleichheit muss ersetzt werden durch Test auf „ausreichend nahe beieinander“**

Der Skalenunterschied zwischen dem größeren Summanden und dem kleineren Summanden kann so klein sein, dass die Summe sich kaum oder gar nicht vom größeren Summanden unterscheidet. Addiert man sehr viele zu kleine Zahlen auf den größeren Summanden, so ändert sich nie etwas, obwohl sich die Summe doch nach einer gewissen Anzahl von Additionen eigentlich signifikant vom größeren Summanden unterscheiden sollte.

Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

Probleme mit Ungenauigkeiten (Beispiele):

- **Umkehrrechnungen liefern nicht genau den Ausgangswert**
- **Bei Addition extrem unterschiedlich großer Zahlen geht die kleinere unter**
- **Subtraktion fast gleich großer Zahlen kann mglw. 0 oder nur inkorrekte Bits ergeben**
- **Test auf Gleichheit muss ersetzt werden durch Test auf „ausreichend nahe beieinander“**

Zwei Zahlen können so nah beieinander liegen, dass ihre Differenz zu klein ist, um mit einem Wert ungleich 0 dargestellt zu werden. Aber auch wenn das Ergebnis der Subtraktion ein Bitmuster ungleich 0 ist, ist es in der Größenordnung der am wenigsten signifikanten Bits, und diese können durch vorherige ungenauere Rechnungen völlig falsch sein.

Gebrochene Zahlen: Typen

TECHNISCHE UNIVERSITÄT DARMSTADT

Probleme mit Ungenauigkeiten (Beispiele):

- **Umkehrrechnungen liefern nicht genau den Ausgangswert**
- **Bei Addition extrem unterschiedlich großer Zahlen geht die kleinere unter**
- **Subtraktion fast gleich großer Zahlen kann mglw. 0 oder nur inkorrekte Bits ergeben**
- **Test auf Gleichheit muss ersetzt werden durch Test auf „ausreichend nahe beieinander“**

Wie das Beispiel oben mit den Quadratwurzeln zeigt, macht es im Allgemeinen wenig Sinn, zwei gebrochenzahlige Werte ganz normal mit dem Vergleichsoperator auf Gleichheit zu testen. Stattdessen muss man sich einen maximalen Differenzwert vorgeben und sieht zwei Zahlen als gleich an, wenn ihr Unterschied diesen Grenzwert nicht übersteigt, das ist der *absolute* Fehler. In der Numerik nimmt man meist den *relativen* Fehler, also die Differenz geteilt durch einen der beiden Werte.

In jedem Fall ergibt sich eine prinzipiell nicht auflösbare Unsicherheit dadurch, dass es keinen naturgesetzlich richtigen maximalen Differenzwert gibt, bis zu dem zwei verschiedene Zahlen als identisch angesehen werden sollen, die Festlegung dieses Wertes ist letztendlich willkürlich und kann sich nachträglich als zu eng oder zu weit herausstellen.

Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

Maximaler/minimaler positiver double-Wert:

- **Double.MAX_VALUE**
- **Double.MIN_VALUE**

Symbolischer Unendlichwert:

- **Double.POSITIVE_INFINITY**
- **Double.NEGATIVE_INFINITY**

Ergebnis Division 0.0 durch 0.0:

- **Double.NaN**

➔ Analog float (Float)

Auch für gebrochenzahlige Datentypen sind Konstanten vordefiniert, wir besprechen hier nicht alle.

Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

Maximaler/minimaler positiver double-Wert:

Double.MAX_VALUE

Double.MIN_VALUE

Symbolischer Unendlichwert:

Double.POSITIVE_INFINITY

Double.NEGATIVE_INFINITY

Ergebnis Division 0.0 durch 0.0:

Double.NaN

➔ Analog float (Float)

Die Konstante MAX_VALUE hat dieselbe Bedeutung wie bei ganzzahligen Datentypen: größte darstellbare Zahl.

Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

Maximaler/minimaler positiver double-Wert:

- Double.MAX_VALUE
- Double.MIN_VALUE

Symbolischer Unendlichwert:

- Double.POSITIVE_INFINITY
- Double.NEGATIVE_INFINITY

Ergebnis Division 0.0 durch 0.0:

- Double.NaN

➔ Analog float (Float)

Die Konstante MIN_VALUE hingegen hat hier eine andere Bedeutung als bei ganzzahligen Datentypen: Bei gebrochenzahligen Datentypen ist MIN_VALUE die kleinste positive darstellbare Zahl. Im Gegensatz zu ganzzahligen Datentypen ist der Wertebereich bei gebrochenzahligen Datentypen symmetrisch um 0 herum, das heißt, zu jeder positiven Zahl gibt es die negative Zahl und umgekehrt Daher wird eine eigene Konstante für die kleinste darstellbare Zahl nicht benötigt, sie ist gerade minus die größte darstellbare Zahl.

Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

Maximaler/minimaler positiver double-Wert:

Double.MAX_VALUE

Double.MIN_VALUE

Symbolischer Unendlichwert:

Double.POSITIVE_INFINITY

Double.NEGATIVE_INFINITY

Ergebnis Division 0.0 durch 0.0:

Double.NaN

➔ Analog float (Float)

Plus und minus unendlich sind ebenfalls schon als Konstanten vordefiniert. Teilt man eine positive Zahl durch 0, kommt plus unendlich heraus, analog minus unendlich bei einer negativen Zahl.

In ganzzahliger Arithmetik führt Division durch 0 hingegen zu Fehlermeldung und Abbruch der Ausführung des Programms.

Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

Maximaler/minimaler positiver double-Wert:

- Double.MAX_VALUE
- Double.MIN_VALUE

Symbolischer Unendlichwert:

- Double.POSITIVE_INFINITY
- Double.NEGATIVE_INFINITY

Ergebnis Division 0.0 durch 0.0:

- Double.NaN

➔ Analog float (Float)

Wenn man hingegen 0 durch 0 als double-Wert teilt oder beispielsweise unendlich von unendlich abzieht, bekommt man einen sehr speziellen double-Wert heraus: NaN für „not a number“. *Nebenbemerkung*: NaN mit dieser Groß- und Kleinschreibung hat sich lange vor Java und weit über Java hinaus als Kürzel für „not a number“ etabliert. Bei der Entwicklung der Klasse Double in Java hat man sich dafür entschieden, die Groß- und Kleinschreibung von NaN wie allgemein üblich festzulegen, auch wenn das im Widerspruch dazu steht, dass in Java alle Buchstaben im Namen einer Konstanten groß sein sollen.

Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

Maximaler/minimaler positiver double-Wert:

- **Double.MAX_VALUE**
- **Double.MIN_VALUE**

Symbolischer Unendlichwert:

- **Double.POSITIVE_INFINITY**
- **Double.NEGATIVE_INFINITY**

Ergebnis Division 0.0 durch 0.0:

- **Double.NaN**

➔ Analog float (Float)

Und das Ganze gibt es natürlich auch für float. Damit schließen wir gebrochenzahlige Datentypen ab ...

Logiktyp boolean: Literale

TECHNISCHE UNIVERSITÄT DARMSTADT

Hat nur zwei verschiedene Literale:

- true
- false

... und kommen nun zum Logiktyp boolean. Logik ist in Java binär, das heißt, es gibt nur die Wahrheitswerte wahr und falsch, englisch true und false. Eine Variable oder Konstante vom Typ boolean wird im Deutschen *boolesch* genannt. Ebenso sind true und false die beiden *booleschen* Literale.

Logiktyp boolean: Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

a	b	!a	a && b	a || b
false	false	true	false	false
false	true	true	false	true
true	false	false	false	true
true	true	false	true	true

Die Logikoperationen lassen sich am Übersichtlichsten durch eine Wahrheitstafel definieren.

Logiktyp boolean: Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

a	b	!a	a && b	a || b
false	false	true	false	false
false	true	true	false	true
true	false	false	false	true
true	true	false	true	true

Zwei boolesche Variable, a und b, können vier verschiedene Kombinationen von Wahrheitswerten haben: false-false, false-true, true-false und true-true.

Logiktyp boolean: Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

a	b	!a	a && b	a || b
false	false	true	false	false
false	true	true	false	true
true	false	false	false	true
true	true	false	true	true

Die Negation wird durch ein vorangestelltes Ausrufezeichen angezeigt. Immer wenn a true ist, ist a negiert false und umgekehrt.

Logiktyp boolean: Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

a	b	!a	a && b	a || b
false	false	true	false	false
false	true	true	false	true
true	false	false	false	true
true	true	false	true	true

Das doppelte Kaufmanns-Und steht für logisches Und, das heißt, dieser Ausdruck ist true genau dann, wenn sowohl a als auch b true sind.

Logiktyp boolean: Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

a	b	!a	a && b	a || b
false	false	true	false	false
false	true	true	false	true
true	false	false	false	true
true	true	false	true	true

Der doppelte vertikale Strich steht für Inklusiv-Oder, das heißt, *dieser* Ausdruck ist true genau dann, wenn entweder a true ist oder b true ist oder wenn beide true sind.

Logiktyp boolean: Ausdrücke

TECHNISCHE UNIVERSITÄT DARMSTADT

```
.........
boolean b1 = ( n > 0 );
boolean b2 = ( x == y ) && ! b1;
b1 = b1 || b2;
.........
if ( b1 == b2 )
.........
while ( b1 != b2 )
.........
```

Diese rein illustrativen booleschen Ausdrücke sollten zeigen, wie man mit diesen Operatoren umgeht.

Logiktyp boolean: Ausdrücke

TECHNISCHE UNIVERSITÄT DARMSTADT

```
.........
boolean b1 = ( n > 0 );
boolean b2 = ( x == y ) && ! b1;
b1 = b1 || b2;
.........
if ( b1 == b2 )
.........
while ( b1 != b2 )
.........
```

Das doppelte Gleichheitszeichen liefert genau dann true zurück, wenn beide Operanden links und rechts denselben Wert haben, und genau dann liefert „!=“ den Wert false zurück. Diesen beiden binären booleschen Operatoren realisieren in Java also den Test auf Gleichheit beziehungsweise Ungleichheit.

***Vorgriff*: Weiter hinten in diesem Kapitel, im Abschnitt zu Operatoren, werden wir diese und verwandte Operatoren noch einmal systematisch betrachten.**

Zeichentyp char: Beispiele

TECHNISCHE UNIVERSITÄT DARMSTADT

```
char c1 = ´a´;        char c7  = ´!´;
char c2 = ´z´;        char c8  = ´?´;
char c3 = ´A´;        char c9  = ´+´;
char c4 = ´Z´;        char c10 = ´%´;
char c5 = ´0´;        char c11 = ´,´;
char c6 = ´9´;        char c12 = ´:´;
```

Zum Schluss der Datentyp für Schriftzeichen, char für englisch character. Wie man an diesen Beispielen schon sieht, setzt man ein konkretes Zeichen in Einzelhochkommas.

Zeichentyp char: Beispiele

TECHNISCHE UNIVERSITÄT DARMSTADT

```
char c1 = ´a´;
char c2 = ´z´;
char c3 = ´A´;
char c4 = ´Z´;
char c5 = ´0´;
char c6 = ´9´;
char c7  = ´!´;
char c8  = ´?´;
char c9  = ´+´;
char c10 = ´%´;
char c11 = ´,´;
char c12 = ´:´;
```

Die Kleinbuchstaben von klein-a bis klein-z.

Zeichentyp char: Beispiele

TECHNISCHE UNIVERSITÄT DARMSTADT

```
char c1 = ´a´;        char c7  = ´!´;
char c2 = ´z´;        char c8  = ´?´;
char c3 = ´A´;        char c9  = ´+´;
char c4 = ´Z´;        char c10 = ´%´;
char c5 = ´0´;        char c11 = ´,´;
char c6 = ´9´;        char c12 = ´:´;
```

Die Großbuchstaben von groß-A bis groß-Z.

Zeichentyp char: Beispiele

TECHNISCHE UNIVERSITÄT DARMSTADT

```
char c1 = ´a´;
char c2 = ´z´;
char c3 = ´A´;
char c4 = ´Z´;
char c5 = ´0´;
char c6 = ´9´;
```

```
char c7  = ´!´;
char c8  = ´?´;
char c9  = ´+´;
char c10 = ´%´;
char c11 = ´,´;
char c12 = ´:´;
```

Die Ziffern von 0 bis 9.

Zeichentyp char: Beispiele

TECHNISCHE UNIVERSITÄT DARMSTADT

```
char c1 = ´a´;
char c2 = ´z´;
char c3 = ´A´;
char c4 = ´Z´;
char c5 = ´0´;
char c6 = ´9´;
```

```
char c7  = ´!´;
char c8  = ´?´;
char c9  = ´+´;
char c10 = ´%´;
char c11 = ´,´;
char c12 = ´:´;
```

Sowie Interpunktionszeichen und Sonderzeichen.

Zeichentyp char: Unicode

TECHNISCHE UNIVERSITÄT DARMSTADT

´a´ ... ´z´	97 ... 122	´:´	58
´A´ ... ´Z´	65 ... 90	´ä´	228
´0´ ... ´9´	48 ... 57	´ö´	246
´!´	33	´ü´	252
´?´	63	´ß´	223
´+´	43	´Ä´	196
´%´	37	´Ö´	214
´,´	44	´Ü´	220

Zeichen werden intern als Zahlen kodiert. Für Java ist Unicode als Kodierung festgelegt. Das sind nichtnegative ganze Zahlen mit 16 Bit, also kleinster Wert 0, größter Wert eins weniger als 2 hoch 16.

Zeichentyp char: Unicode

TECHNISCHE UNIVERSITÄT DARMSTADT

´a´ ... ´z´	97 ... 122	´:´	58
´A´ ... ´Z´	65 ... 90	´ä´	228
´0´ ... ´9´	48 ... 57	´ö´	246
´!´	33	´ü´	252
´?´	63	´ß´	223
´+´	43	´Ä´	196
´%´	37	´Ö´	214
´,´	44	´Ü´	220

Für Kleinbuchstaben, Großbuchstaben und Ziffern gibt es jeweils Bereiche. Zum Beispiel hat klein-a den Wert 97, klein-b hat Wert 98, klein-c hat Wert 99 und so weiter, und klein-z hat schließlich Wert 122. Analog Großbuchstaben und Ziffern, beginnend mit 65 beziehungsweise 48.

Zeichentyp char: Unicode

TECHNISCHE UNIVERSITÄT DARMSTADT

´a´ ... ´z´	97 ... 122	´:´	58
´A´ ... ´Z´	65 ... 90	´ä´	228
´0´ ... ´9´	48 ... 57	´ö´	246
´!´	33	´ü´	252
´?´	63	´ß´	223
´+´	43	´Ä´	196
´%´	37	´Ö´	214
´,´	44	´Ü´	220

Die Interpunktionszeichen und Sonderzeichen haben jeweils einzelne Werte ohne Zusammenhang untereinander. Sie liegen alle im Bereich von 0 bis 127, weil sie schon im ASCII-Zeichensatz vorkommen, der als die ersten 128 Zeichen in Unicode übernommen wurde.

Zeichentyp char: Unicode

TECHNISCHE UNIVERSITÄT DARMSTADT

´a´ ... ´z´	97 ... 122	´:´	58
´A´ ... ´Z´	65 ... 90	´ä´	228
´0´ ... ´9´	48 ... 57	´ö´	246
´!´	33	´ü´	252
´?´	63	´ß´	223
´+´	43	´Ä´	196
´%´	37	´Ö´	214
´,´	44	´Ü´	220

Die Umlaute und das scharfe s haben ebenfalls jeweils einzelne Werte. Sie haben alle Werte zwischen 128 und 255, denn sie gehören als Zeichen nichtangelsächsischen Ursprungs nicht zum ASCII-Zeichensatz, sondern zum ebenfalls übernommenen Zeichensatz ISO-Latin-1, der neben den deutschen Sonderzeichen auch Sonderzeichen aus romanischen Sprachen enthält.

Zeichentyp char: Sonderzeichen

TECHNISCHE UNIVERSITÄT DARMSTADT

Zeichen	Bedeutung	Unicode (dezimal)
´\t´	Horizontaler Tab	9
´\b´	Backspace	8
´\n´	Neue Zeile	10
´\´´	´	39
´\“´	“	34
´\\´	\	92

Für einzelne wichtige Zeichen gibt es Sonderschreibweisen.

Zeichentyp char: Sonderzeichen

TECHNISCHE UNIVERSITÄT DARMSTADT

Zeichen	Bedeutung	Unicode (dezimal)
´\t´	Horizontaler Tab	9
´\b´	Backspace	8
´\n´	Neue Zeile	10
´\´´	´	39
´\“´	“	34
´\\´	\	92

Zum einen sind das Zeichen zur Textformatierung wie Tabulator und newline beziehungsweise Zeichen zur Textmodifikation wie Backspace, also Rückwärtslöschen.

Zeichentyp char: Sonderzeichen

TECHNISCHE UNIVERSITÄT DARMSTADT

Zeichen	Bedeutung	Unicode (dezimal)
´\t´	Horizontaler Tab	9
´\b´	Backspace	8
´\n´	Neue Zeile	10
´\´´	´	39
´\“´	“	34
´\\´	\	92

Zum anderen sind das Zeichen, die für die Darstellung anderer Schriftzeichen verwendet werden. Wenn man eines dieser Zeichen nicht zur Darstellung anderer Zeichen verwenden will, sondern wirklich dieses Zeichen meint, muss man es so umschreiben.

Zeichentyp char: hexadezimal

TECHNISCHE UNIVERSITÄT DARMSTADT

- Auch mit Nummer im Hexadezimalcode
- Hexadezimale Ziffern:

  0 1 2 3 4 5 6 7 8 9 A B C D E F

- Beispiel: ´\u03A9´ (dezimal 937) ist ´Ω´

Nun umfasst Unicode etliche tausend Zeichen. Für die alle gibt es natürlich nicht solche Umschreibungen.

Zeichentyp char: hexadezimal

TECHNISCHE UNIVERSITÄT DARMSTADT

- Auch mit Nummer im Hexadezimalcode
- Hexadezimale Ziffern:

  0 1 2 3 4 5 6 7 8 9 A B C D E F

- Beispiel: ´\u03A9´ (dezimal 937) ist ´Ω´

Stattdessen kann man jedes Unicode-Zeichen auch durch Angabe seiner Nummer schreiben, allerdings nicht im üblichen Dezimalsystem, sondern im Hexadezimalsystem. Das heißt, die Basis ist nicht 10, sondern 16.

Zeichentyp char: hexadezimal

TECHNISCHE UNIVERSITÄT DARMSTADT

- Auch mit Nummer im Hexadezimalcode
- Hexadezimale Ziffern:

  0 1 2 3 4 5 6 7 8 9 A B C D E F

- Beispiel: ´\u03A9´ (dezimal 937) ist ´Ω´

Die ersten zehn Ziffern zur Darstellung von Hexadezimalzahlen in Java sind die üblichen.

Zeichentyp char: hexadezimal

TECHNISCHE UNIVERSITÄT DARMSTADT

- Auch mit Nummer im Hexadezimalcode
- Hexadezimale Ziffern:

  0 1 2 3 4 5 6 7 8 9 A B C D E F

- Beispiel: ´\u03A9´ (dezimal 937) ist ´Ω´

Hinzu kommen sechs Ziffern für die dezimalen Zahlenwerte 10, 11, 12, 13 14 und 15. Diese sechs Ziffern werden mit den ersten sechs Großbuchstaben im Alphabet dargestellt.

Zeichentyp char: hexadezimal

TECHNISCHE UNIVERSITÄT DARMSTADT

- Auch mit Nummer im Hexadezimalcode
- Hexadezimale Ziffern:

  0 1 2 3 4 5 6 7 8 9 A B C D E F

- Beispiel: ´\u03A9´ (dezimal 937) ist ´Ω´

Insgesamt also 16 Ziffern mit den dezimalen Zahlenwerten 0 bis 15.

Zeichentyp char: hexadezimal

TECHNISCHE UNIVERSITÄT DARMSTADT

- Auch mit Nummer im Hexadezimalcode
- Hexadezimale Ziffern:

  0 1 2 3 4 5 6 7 8 9 A B C D E F

- Beispiel: ´\u03A9´ (dezimal 937) ist ´Ω´

So kann man nun ein Zeichen mit Unicode-Nummer schreiben. Wir gehen die Bestandteile im Einzelnen durch.

Zeichentyp char: hexadezimal

TECHNISCHE UNIVERSITÄT DARMSTADT

- Auch mit Nummer im Hexadezimalcode
- Hexadezimale Ziffern:

  0 1 2 3 4 5 6 7 8 9 A B C D E F

- Beispiel: ´\u03A9´ (dezimal 937) ist ´Ω´

Wie bisher wird das Zeichen in einzelne Hochkommas gesetzt.

Zeichentyp char: hexadezimal

TECHNISCHE UNIVERSITÄT DARMSTADT

- Auch mit Nummer im Hexadezimalcode
- Hexadezimale Ziffern:

  0 1 2 3 4 5 6 7 8 9 A B C D E F

- Beispiel: ´\u03A9´ (dezimal 937) ist ´Ω´

Der Rückwärtsschrägstrich, englisch Backslash vorneweg wie bei den Sonderzeichen.

Zeichentyp char: hexadezimal

TECHNISCHE UNIVERSITÄT DARMSTADT

- Auch mit Nummer im Hexadezimalcode
- Hexadezimale Ziffern:

  0 1 2 3 4 5 6 7 8 9 A B C D E F

- Beispiel: ´\u03A9´ (dezimal 937) ist ´Ω´

Gefolgt von einem kleinen u, um anzuzeigen, dass jetzt eine Unicode-Nummer kommt.

Zeichentyp char: hexadezimal

TECHNISCHE UNIVERSITÄT DARMSTADT

- Auch mit Nummer im Hexadezimalcode
- Hexadezimale Ziffern:

  0 1 2 3 4 5 6 7 8 9 A B C D E F

- Beispiel: ´\u03A9´ (dezimal 937) ist ´Ω´

Und das ist jetzt die Unicode-Nummer, dargestellt durch vier hexadezimale Stellen, was genau den 16 Bit des Datentyps char entspricht.

Zeichentyp char: hexadezimal

TECHNISCHE UNIVERSITÄT DARMSTADT

- Auch mit Nummer im Hexadezimalcode
- Hexadezimale Ziffern:

  0 1 2 3 4 5 6 7 8 9 A B C D E F

- Beispiel: ´\u03A9´ (dezimal 937) ist ´Ω´

Im Dezimalcode ist das die 937.

Zeichentyp char: hexadezimal

TECHNISCHE UNIVERSITÄT DARMSTADT

- Auch mit Nummer im Hexadezimalcode
- Hexadezimale Ziffern:

  0 1 2 3 4 5 6 7 8 9 A B C D E F

- Beispiel: ´\u03A9´ (dezimal 937) ist ´Ω´

Und diese Nummer steht für Großbuchstabe Omega.

Unicode-Tabellen geben die Nummern der Zeichen typischerweise schon als Hexadezimalzahlen an, so dass man die Hexadezimalnummer nicht aus dem Dezimalsystem umrechnen muss, sondern einfach als Hexadezimalzahl nachschlagen kann.

TECHNISCHE UNIVERSITÄT DARMSTADT

Arithmetische Operatoren

und

Vergleichsoperatoren

Oracle Java Tutorials: Operators
(ohne Abschnitt Bitwise and Bit Shift)

Wir haben schon einige Operationen auf Zahlen gesehen. Als nächstes schauen wir uns dieses Thema systematisch an.

Beispiele

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int i = 2 * ( 3 + 5 ) % ( 7 – 3 ) + 4;

int j = 8 – i * i;

int k = –123;

k = +123;

i = ( 3 + j ) * ( i – 5 );
```

Hier sehen Sie ein paar Beispiele für mathematische Ausdrücke, die diverse arithmetische Operationen enthalten.

Beispiele

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int i = 2 * ( 3 + 5 ) % ( 7 – 3 ) + 4;
int j = 8 – i * i;
int k = –123;
k = +123;
i = ( 3 + j ) * ( i – 5 );
```

Einerseits können Variable durch beliebig komplexe mathematische Ausdrücke initialisiert werden.

Beispiele

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int i = 2 * ( 3 + 5 ) % ( 7 – 3 ) + 4;
int j = 8 – i * i;
int k = –123;
k = +123;
i = ( 3 + j ) * ( i – 5 );
```

Andererseits kann man den Wert einer solchen Variable danach beliebig oft mit unterschiedlichen Werten überschrieben werden, die ebenfalls Werte von beliebig komplexen mathematischen Ausdrücken sein können.

Beispiele

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int i = 2 * ( 3 + 5 ) % ( 7 – 3 ) + 4;
int j = 8 – i * i;
int k = –123;
k = +123;
i = ( 3 + j ) * ( i – 5 );
```

Hier sehen Sie ein Beispiel dafür, dass der Wert einer Variable in einem Ausdruck sein kann, der eben diesen Wert überschreibt. Zuerst wird der bisherige Wert von i ausgelesen und zusammen mit dem Wert von j verwendet, um den Wert des Gesamtausdrucks auf der rechten Seite des Zuweisungszeichens zu berechnen. Dieser Wert wird dann i zugewiesen.

Arithmetische Kurzformen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
i = i + 5;
i += 5;

i = i – 3;
i –= 3;

i = i * 7;
i *= 7;

i = i / 4;
i /= 4;

i = i % 6;
i %= 6;

i = i + 1;
i += 1;
i++;

i = i – 1;
i –= 1;
i – – ;
```

Für einige häufige Fälle gibt es in Java und einigen anderen Sprachen auch Kurzformen.

Arithmetische Kurzformen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
i = i + 5;
i += 5;

i = i – 3;
i –= 3;

i = i * 7;
i *= 7;

i = i / 4;
i /= 4;
```

```
i = i % 6;
i %= 6;

i = i + 1;
i += 1;
i++;

i = i – 1;
i –= 1;
i – – ;
```

Konkret geht es um den häufigen Fall, dass der Wert in einem Objekt mit einem anderen Wert verknüpft und das Ergebnis dem Objekt wieder zugewiesen wird.

Arithmetische Kurzformen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
i = i + 5;
i += 5;

i = i – 3;
i –= 3;

i = i * 7;
i *= 7;

i = i / 4;
i /= 4;

i = i % 6;
i %= 6;

i = i + 1;
i += 1;
i++;

i = i – 1;
i –= 1;
i – – ;
```

Wie in jedem dieser Fälle die Kurzform gebildet wird, sollte selbsterklärend sein.

Arithmetische Kurzformen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
i = i + 5;
i += 5;

i = i – 3;
i –= 3;

i = i * 7;
i *= 7;

i = i / 4;
i /= 4;

i = i % 6;
i %= 6;

i = i + 1;
i += 1;
i++;

i = i – 1;
i –= 1;
i – – ;
```

Für den besonders häufigen Fall, dass ein Wert um 1 erhöht oder vermindert wird, gibt es jeweils eine noch kürzere Form, die wir schon bei for-Schleifen gesehen haben. Dieser Fall – Erhöhung beziehungsweise Verminderung um 1 – heißt Inkrement beziehungsweise Dekrement.

Operatoren

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele:

+	−	*	/	%	
	+	−			
	++	−−			
=	+=	−=	*=	/=	%=
==	!=	<	>	<=	>=

Hier sehen Sie eine Übersicht über die wichtigsten Operatoren auf Zahlen in Java.

Operatoren

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele:

+	−	*	/	%	
	+	−			
	++	−−			
=	+=	−=	*=	/=	%=
==	!=	<	>	<=	>=

Die vier Grundrechenarten plus Modulobildung sind binäre Infix-Operatoren. *Binär* heißt, dass jeder von ihnen zwei Operanden hat, und *Infix* heißt, dass der Operator zwischen den beiden Operanden steht.

Operatoren

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele:

+	−	*	/	%	
	+		−		
	++		−−		
=	+=	−=	*=	/=	%=
==	!=	<	>	<=	>=

Die beiden Vorzeichen sind hingegen *unäre Präfix*-Operatoren. *Unär* im Gegensatz zu *binär* heißt, dass es nur einen Operanden gibt, und *Präfix* statt *Infix* heißt, dass der Operator dem Operanden vorangestellt ist.

Operatoren

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele:

+	−	*	/	%	
	+	−			
	++	−−			
=	+=	−=	*=	/=	%=
==	!=	<	>	<=	>=

Inkrement- und Dekrement-Operator sind unär. Wir haben sie als Postfix-Operatoren kennengelernt, also dem Operanden nachgestellt. Man kann sie statt dessen auch dem Operator voranstellen. Warum es beide Möglichkeiten gibt und was der Unterschied ist, werden wir später sehen.

Operatoren

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele:

+	−	*	/	%	
	+		−		
	++		−−		
=	+=	−=	*=	/=	%=
==	!=	<	>	<=	>=

Alle diese Operatoren haben wir ebenfalls schon gesehen. Man nennt sie die *zuweisungsbasierten* Operatoren.

Operatoren

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele:

+	−	*	/	%	
	+	−			
	++	−−			
=	+=	−=	*=	/=	%=
==	!=	<	>	<=	>=

Und das sind die Vergleichsoperatoren, von links nach rechts: Test auf gleich, ungleich, kleiner, größer, kleiner oder gleich, größer oder gleich.

TECHNISCHE UNIVERSITÄT DARMSTADT

Arithmetische Ergebnistypen

Bei arithmetischen Ausdrücken müssen wir genau auseinanderhalten, was jeweils die Ergebnistypen sind. Dazu gibt es einige Regeln, die wir uns im Folgenden anschauen.

Arithmetische Ergebnistypen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
byte   b;
char   c;
short  s;
int    i;
long   l;
float  f;
double d;
```

***Erinnerung*: Das sind die einzelnen arithmetischen Datentypen in Java.**

Arithmetische Ergebnistypen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
byte   b;
char   c;
short  s;
int    i;
long   l;
float  f;
double d;
```

Wobei auch char tatsächlich ein Zahlentyp ist. Die Verbindung eines Zahlenwertes im Datentyp char mit einem Zeichen ergibt sich erst auf einer anderen Ebene, nämlich weil beispielsweise Methoden zum Schreiben eines Zeichens auf den Bildschirm oder in eine Datei etwas ganz anderes mit dem Zahlenwert machen als bei den anderen ganzzahligen Typen: nicht den Zahlenwert dezimal auf dem Bildschirm ausgeben, sondern statt dessen das einzelne Zeichen, das in Unicode dieser Zahl zugeordnet ist.

Arithmetische Ergebnistypen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
byte   b;
char   c;
short  s;
int    i;
long   l;
float  f;
double d;
```

Wenn im Folgenden die Addition gezeigt wird, dann ist das jeweils nur beispielhaft. Alles Gesagte gilt jeweils völlig identisch auch für die anderen arithmetischen Operationen, also Subtraktion, Multiplikation, Division und Modulo.

Arithmetische Ergebnistypen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
byte   b;
char   c;
short  s;
int    i;
long   l;
float  f;
double d;
```

Ergebnistyp int:

```
b+b
c+c
s+s
i+i
```

Erste Erkenntnis: Wenn eine arithmetische Operation zwei Werte vom Typ byte, char, short oder int als Operanden hat, dann ist das Ergebnis der arithmetischen Operation int.

Arithmetische Ergebnistypen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
byte   b;
char   c;
short  s;
int    i;
long   l;
float  f;
double d;
```

Ergebnistyp int:

b+b	b+c c+b
c+c	b+s s+b
s+s	b+i i+b
i+i	c+s s+c
	c+i i+c
	s+i i+s

Aber auch bei jeder Kombination von Operanden aus diesen vier Datentypen ist das Ergebnis int, egal von welchem Typ genau der erste und von welchem Typ genau der zweite Operand ist.

Sind beide Operanden einer Division von ganzzahligen Typen, dann wird ganzzahlige Division mit Rest angewandt.

***Nebenbemerkung*: In bestimmten, sehr einfachen Situationen, in denen der Compiler abprüfen kann, dass das Ergebnis einer arithmetischen Operation mit char im Bereich char bleibt, ist das Ergebnis wieder char. Aber uns geht es hier um die Regeln, nicht um die Ausnahmen.**

Arithmetische Ergebnistypen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
byte   b;
char   c;
short  s;
int    i;
long   l;
float  f;
double d;
```

Ergebnistyp long:

b+l	l+b
c+l	l+c
s+l	l+s
i+l	l+i
l+l	

Ist einer der beiden ganzzahligen Operanden hingegen vom Typ long und der andere von einem der Typen byte, char, short, int oder long, dann ist das Ergebnis der arithmetischen Operation vom Typ long.

Auch hier ist die Division ganzzahlig mit Rest.

Arithmetische Ergebnistypen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
byte   b;
char   c;
short  s;
int    i;
long   l;
float  f;
double d;
```

Ergebnistyp float:

b+f f+b
c+f f+c
s+f f+s
i+f f+i
l+f f+l
f+f

Jetzt nehmen wir float hinzu. Ist der andere Operand von einem der Typen byte, char, short, int, long oder float, dann ist das Ergebnis der arithmetischen Operation vom Datentyp float.

Ist einer der beiden Operanden einer Division vom Typ float, dann wird gebrochenzahlige Division angewandt.

Arithmetische Ergebnistypen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
byte   b;
char   c;
short  s;
int    i;
long   l;
float  f;
double d;
```

Ergebnistyp double:

b+d d+b
c+d d+c
s+d d+s
i+d d+i
l+d d+l
f+d d+f
d+d

Schlussendlich, ist einer der beiden Operanden vom Typ double, dann ist das Ergebnis der arithmetischen Operation auf jeden Fall vom Typ double.

Auch in diesem Fall wird gebrochenzahlige Division angewandt.

TECHNISCHE UNIVERSITÄT DARMSTADT

Konversionen

Arithmetische Datentypen können auch bei Wertzuweisungen und ähnlichem ineinander konvertiert werden.

Konversionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
byte   b;
char   c;
short  s;
int    i;
long   l;
float  f;
double d;
```

Links sehen Sie wieder wie eben die arithmetischen Datentypen.

Konversionen: implizite

TECHNISCHE UNIVERSITÄT DARMSTADT

```
byte   b;
char   c;
short  s;
int    i;
long   l;
float  f;
double d;
```

```
s = b;   f = c;   f = i;
i = b;   d = c;   d = i;
l = b;   i = s;   f = l;
f = b;   l = s;   d = l;
d = b;   f = s;   d = f;
i = c;   d = s;
l = c;   l = i;
```

Die hier gezeigten sind die sicheren Fälle, das heißt, man kann sich sicher sein, dass die beiden Variablen links und rechts des Zuweisungszeichens gemäß den Auswertungsregeln ihrer jeweiligen Datentypen genau denselben Zahlenwert beschreiben.

Die sicheren Fälle sind genau die, bei denen links des Gleichheitszeichens ein Typ steht, der in der Liste der Typen weiter unten steht wie der Typ rechts vom Gleichheitszeichen.

Es gibt da nur eine Ausnahme: Die Konversion von char nach short ist nicht ganz sicher und deshalb hier nicht aufgeführt, obwohl die Zuweisung s = c durch den Compiler ginge.

***Nebenbemerkung*: Das liegt daran, dass short und char dieselbe Anzahl Bits haben, aber der Wertebereich von short enthält positive und negative Zahlen (und natürlich die 0), während char nur nichtnegative Zahlen kennt. Ungefähr die Hälfte aller Werte im Wertebereich von char kommt daher im Datentyp short als negative Zahl an, und das kann nach menschlichem Ermessen praktisch nie das eigentlich intendierte Ergebnis sein.**

Konversionen: implizite

TECHNISCHE UNIVERSITÄT DARMSTADT

```
byte   b;          s = b;  f = c;  f = i;
char   c;          i = b;  d = c;  d = i;
short  s;          l = b;  i = s;  f = l;
int    i;          f = b;  l = s;  d = l;
long   l;          d = b;  f = s;  d = f;
float  f;          i = c;  d = s;
double d;          l = c;  l = i;
```

Zu beachten ist, dass es in diesen drei Fällen zu kleinen Ungenauigkeiten bei der Konversion des Wertes kommen kann. Nicht jeder sehr große int-Wert ist exakt in float darstellbar, und nicht jeder sehr große long-Wert ist exakt in float oder double darstellbar. Normalerweise merkt man davon nichts, weil entweder die Zahlen, um die es geht, nicht so groß sind oder die kleine Ungenauigkeit keine Konsequenzen hat. Daher zählen diese drei Konversionen trotz dieser Ungenauigkeit zu den sicheren.

Konversionen: explizite

TECHNISCHE UNIVERSITÄT DARMSTADT

```
byte   b;
char   c;
short  s;
int    i;
long   l;
float  f;
double d;
```

Beispiele:

```
c = (char)i;
b = (byte)i;
s = (short)l;
i  = (int)l;
f  = (float)d;
```

Man kann aber auch in den anderen, den unsicheren Fällen konvertieren. Das Problem ist, dass sehr viele Werte des Ausgangstyps nicht im Zieltyp darstellbar sind. Steht ein solcher nicht darstellbarer Wert auf der rechten Seite der Zuweisung, dann kommt auf der linken Seite ein Murkswert heraus.

Konversionen: explizite

TECHNISCHE UNIVERSITÄT DARMSTADT

```
byte   b;
char   c;
short  s;
int    i;
long   l;
float  f;
double d;
```

Beispiele:

```
c = (char)i;
b = (byte)i;
s = (short)l;
i = (int)l;
f = (float)d;
```

In einem unsicheren Fall muss man explizit sagen, dass man konvertieren will, indem man den Zieltyp in Klammern angibt wie hier gezeigt. Dies nennt man eine *ex*plizite Konversion im Gegensatz zu den *im*pliziten Konversionen in den sicheren Fällen vorher, wo eben nichts weiter hinzuschreiben war.

Eine erzwungene Chance für den Programmierer, sich noch einmal zu überlegen, ob er wirklich weiß, was er da tut.

Die hier gezeigten Beispiele sind natürlich nur eine Auswahl aller unsicheren Fälle. Unsicher sind alle Fälle, die nicht unter den sicheren auf den letzten Folien waren.

Beispiel explizite Konversion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
final int DIFF = 32;
char lowerA = ´a´;
char lowerZ = ´z´;
char upperA = (char) ( lowerA – DIFF );
char upperZ = (char) ( lowerZ – DIFF );
```

Zum Abschluss dieses Abschnitts noch ein sehr kleines Beispiel dafür, dass auch unsichere Konversionen manchmal sinnvoll sind.

Beispiel explizite Konversion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
final int DIFF = 32;
char lowerA = ´a´;
char lowerZ = ´z´;
char upperA = (char) ( lowerA – DIFF );
char upperZ = (char) ( lowerZ – DIFF );
```

Wir richten zwei Variable vom Typ char ein und initialisieren sie mit den Zeichen klein-a und klein-z.

Beispiel explizite Konversion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
final int DIFF = 32;
char lowerA = ´a´;
char lowerZ = ´z´;
char upperA = (char) ( lowerA – DIFF );
char upperZ = (char) ( lowerZ – DIFF );
```

Dann richten wir noch zwei Variable ein und initialisieren sie mit den Zeichen groß-A und groß-Z. Wie das?

Beispiel explizite Konversion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
final int DIFF = 32;
char lowerA = ´a´;
char lowerZ = ´z´;
char upperA = (char) ( lowerA – DIFF );
char upperZ = (char) ( lowerZ – DIFF );
```

Nun, in Unicode und somit im Datentyp char ist die Nummer jedes Großbuchstabens A bis Z genau um 32 kleiner als die Nummer des zugehörigen Kleinbuchstabens. Wir ziehen also 32 vom Kleinbuchstaben ab und erhalten so den Großbuchstaben.

Beispiel explizite Konversion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
final int DIFF = 32;
char lowerA = ´a´;
char lowerZ = ´z´;
char upperA = (char) ( lowerA – DIFF );
char upperZ = (char) ( lowerZ – DIFF );
```

Aber: Der Ergebnistyp dieser Subtraktion ist int, wie wir soeben festgestellt hatten. Wir wollen aber wieder char herausbekommen. Die Konversion von int nach char ist unsicher, also müssen wir char in Klammern vorneweg schreiben.

In diesem Fallbeispiel wissen wir definitiv, was wir tun. Wir wissen definitiv, dass der richtige Wert und kein Murkswert herauskommt, solange wir sicher sein können, dass der char-Wert, von dem wir 32 abziehen, ein Kleinbuchstabe ist. Also konvertieren wir bedenkenlos.

TECHNISCHE UNIVERSITÄT DARMSTADT

Bedingungsoperator

Neben unären und binären Operatoren gibt es in Java auch einen ternären Operator, also mit drei Operanden.

Bedingungsoperator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int n = .....;
boolean b = .....;
double x = .....;
int m = .....;

double z  =  ( n > 0 ) && b  ?  x  :  ( m + 3 );
```

Hier sehen Sie ein kleines, rein illustratives Beispiel für den Bedingungsoperator.

Bedingungsoperator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int n = .....;
boolean b = .....;
double x = .....;
int m = .....;

double z  =  ( n > 0 ) && b  ?  x  :  ( m + 3 );
```

Speziell hier sehen Sie die drei Operanden.

Bedingungsoperator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int n = .....;
boolean b = .....;
double x = .....;
int m = .....;

double z = ( n > 0 ) && b ? x : ( m + 3 );
```

Der erste Operand muss vom Typ boolean sein, also true oder false zurückliefern.

Bedingungsoperator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int n = .....;
boolean b = .....;
double x = .....;
int m = .....;

double z  =  ( n > 0 ) && b  ?  x  :  ( m + 3 );
```

Die anderen beiden Operanden können im Prinzip erst einmal beliebige Typen haben, wir spezifizieren das gleich genauer. In diesem Beispiel haben beide Ausdrücke den Typ double. Wie Sie sehen, können die beiden Ausdrücke wie üblich atomar sein wie in diesem Beispiel der erste dieser beiden Operanden oder zusammengesetzt wie der zweite dieser beiden Operanden.

Bedingungsoperator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int n = .....;
boolean b = .....;
double x = .....;
int m = .....;

double z = ( n > 0 ) && b ? x : ( m + 3 );
```

Der Wert, der vom Bedingungsoperator zurückgeliefert wird, ist entweder der Wert des zweiten oder der Wert des dritten Operanden, und zwar genau dann der Wert des zweiten Operanden, wenn der erste Operand true liefert.

Jetzt können wir genauer die Typen des zweiten und dritten Operanden spezifizieren: Sowohl der zweite als auch der dritte Operand muss entweder gleich dem Typ sein, der bei der Verwendung der Rückgabe des Bedingungsoperators erwartet wird, oder implizit darin konvertierbar sein. In diesem Beispiel wird double erwartet, weil die Rückgabe in einer Variablen vom Typ double gespeichert werden soll, und der zweite und der dritte Operand sind vom Typ double beziehungsweise int, das passt also.

Bedingungsoperator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
double z  =  ( n > 0 )  ?  x  :  ( y + 3.14 );
```

```
double z;
if ( n > 0 )
   z = x;
else
   z = y + 3.14;
```

Offensichtlich ist der Bedingungsoperator einfach eine Abkürzung für if-Verzweigungen wie die unten gezeigte, die völlig äquivalent zum Ausdruck oben ist. Konkret lässt sich eine if-Verzweigung durch einen Ausdruck mit Bedingungsoperator ersetzen, wenn sie einen else-Teil hat und wenn in beiden Teilen jeweils nichts anderes passiert als dieselbe Variable jeweils auf einen Wert zu setzen.

TECHNISCHE UNIVERSITÄT DARMSTADT

Bindungsstärke von Operatoren

Java Oracle Tutorials: Operators

Wenn ein arithmetischer oder boolescher Ausdruck mehrere verschiedene Operatoren enthält und die Auswertungsreihenfolge nicht durch Klammerung festgelegt ist, stellt sich die Frage, in welcher Reihenfolge die Operatoren nun ausgewertet werden. Denn davon hängt ja das Ergebnis des Ausdrucks massiv ab.

Bindungsstärke

TECHNISCHE UNIVERSITÄT DARMSTADT

```
x * y + z

b == x < y;

b = x == y;

a == b ? c == d : e != f;

a = b ? x : y;
```

Wir schauen uns nur ein paar ausgewählte Beispiele an. Zur Bindungsstärke von Operatoren finden Sie in sicherlich jedem Java-Buch und in unzähligen Internetquellen die Übersichtstabelle. Wichtig ist, dass Bindungsstärke transitiv ist: Wenn Operator A stärker als Operator B und B stärker als C bindet, dann bindet A auch stärker als C. Und wenn A und B gleich stark binden und C stärker beziehungsweise schwächer als A bindet, dann bindet C auch stärker beziehungsweise schwächer als B.

Bindungsstärke

TECHNISCHE UNIVERSITÄT DARMSTADT

```
x * y + z

b == x < y;

b = x == y;

a == b ? c == d : e != f;

a = b ? x : y;
```

Wie in der Schulmathematik, so gilt auch in Java Punkt- vor Strichrechnung. Soll in diesem Beispiel die Addition vor der Multiplikation ausgeführt werden, dann muss y + z in Klammern gesetzt werden.

Bindungsstärke

TECHNISCHE UNIVERSITÄT DARMSTADT

```
x * y + z

b == x < y;

b = x == y;

a == b ? c == d : e != f;

a = b ? x : y;
```

Der Test auf Gleichheit und Ungleichheit, also Operator == und !=, haben geringere Bindungsstärke als die Größenvergleiche, das heißt, kleiner, größer, kleiner-gleich und größer-gleich, so dass b hier ein boolean sein muss und der Wert von b auf Gleichheit mit dem booleschen Ergebnis von x kleiner y getestet wird.

Bindungsstärke

TECHNISCHE UNIVERSITÄT DARMSTADT

```
x * y + z

b == x < y;

b = x == y;

a == b ? c == d : e != f;

a = b ? x : y;
```

Die zuweisungsbasierten Operatoren haben geringere Bindungssstärke als der Test auf Gleichheit. In dieser Zeile muss b ein boolean sein, und das Ergebnis des Tests von x und y auf Gleichheit wird b zugewiesen.

Bindungsstärke

TECHNISCHE UNIVERSITÄT DARMSTADT

```
x * y + z

b == x < y;

b = x == y;

a == b ? c == d : e != f;

a = b ? x : y;
```

Der Bedingungsoperator hat geringere Bindungsstärke als der Test auf Gleichheit beziehungsweise Ungleichheit, ...

Bindungsstärke

TECHNISCHE UNIVERSITÄT DARMSTADT

```
x * y + z

b == x < y;

b = x == y;

a == b ? c == d : e != f;

a = b ? x : y;
```

... aber doch noch höhere als die zuweisungsbasierten Operatoren.

Bindungsstärke

TECHNISCHE UNIVERSITÄT DARMSTADT

Auswertungsreihenfolge:

- **Von links nach rechts: a – b – c – d**
- **Von rechts nach links: a = b = c = d**

Jeder Operator, den man so wie diese beiden zu Ketten verknüpfen kann, hat eine Auswertungsreihenfolge, entweder von links nach rechts oder von rechts nach links. Auch diese Information finden Sie in der Regel in einer Tabelle zur Bindungsstärke der Operatoren.

Bindungsstärke

TECHNISCHE UNIVERSITÄT DARMSTADT

Auswertungsreihenfolge:

- **Von links nach rechts: a – b – c – d**
- **Von rechts nach links: a = b = c = d**

Zum Beispiel die Subtraktion bindet von links nach rechts, das heißt, von a wird zuerst b abgezogen, von der Differenz dann c und schließlich d, das ist das Ergebnis des Gesamtausdrucks.

Bindungsstärke

TECHNISCHE UNIVERSITÄT DARMSTADT

Auswertungsreihenfolge:

- **Von links nach rechts: a – b – c – d**
- **Von rechts nach links: a = b = c = d**

Zuweisungen hingegen werden von rechts nach links ausgewertet: Erst wird c der Wert von d zugewiesen, dann b der Wert von c und schließlich a der Wert von b. Am Ende haben also a, b und c den Wert, den d unmittelbar vor Ausführung dieser Kette von Zuweisungen hatte. Dabei hat d natürlich seinen Wert behalten.