3.5 Typkonvertierungen

Bei einer Wertübertragung von einem einfachen Datentyp in einen anderen sind in Java Regeln zur Typkonvertierung zu beachten. Man unterscheidet **erweiternde** und **einschränkende** Konvertierungen.

Grob gesagt geht es darum, ob bei einer Konvertierung der Wertebereich vergrößert oder verringert wird.

Folgende Tabelle zeigt, welche einfachen Datentypen ineinander konvertiert werden können:

```
byte → short →  int  → long → float → double
                 ↑
               char
```

Konvertierungen in Pfeilrichtung sind erweiternd und Konvertierungen entgegen der Pfeilrichtung sind einschränkend.

Insbesondere sind Konvertierungen mit dem Datentyp **boolean** nicht erlaubt. Einfache Umwandlungen lassen sich hier mit Hilfe des ternären Operators **? :** erzielen.

Erweiternde Konvertierungen werden in Java automatisch vorgenommen:

Grundlagen/Datentypen/CastBsp.java

```
10    //float einzelgewicht = 1.3;     // Fehler: double → float
11    float einzelgewicht = 1.3f;
12    byte  anzahl = 5;
13    float gesamtgewicht = anzahl * einzelgewicht;
14    //                                // anzahl: byte → float
15
16    double PI = 3.1415927;
17    double radius = 2.5;
18    double umfang = 2 * PI * radius; // 2 → 2.0: int → double
```

Einschränkende Konvertierungen müssen in Java explizit mit Hilfe des **Type-Cast-Operators** durchgeführt werden:

```
20    double x = 2.0 * 0.5;
21    int y;
22    //y = x;                          // Fehler: double → int
23    y = (int)x;                       // okay (expliziter Cast)
```

3.6 Operatoren und Ausdrücke

Operatoren führen bestimmte Operationen auf einer, zwei oder drei Variablen aus (**unäre**, **binäre**, **ternäre** Operatoren).

- Unäre Operatoren können in **Präfix**- oder **Postfix**-Notation benutzt werden:

  Präfix-Operator **operator op** z. B. **++i**
  Postfix-Operator **op operator** z. B. **i++**

- Binäre Operatoren benutzen **Infix**-Notation:

  Infix-Operator **op1 operator op2** z. B. **x + y**

- In Java gibt es auch einen ternären Operator:

  expr ? op1 : op2 z. B. **(i > 0) ? 1 : -1**

Man unterteilt die Operatoren in Java in:

- arithmetische Operatoren (**+**, **-**, *****, **/**, **%**; **++**, **--**)
- Vergleichsoperatoren (**>**, **>=**, **<**, **<=**, **==**, **!=**)
- logische Operatoren (**&&**, **||**, **!**, **&**, **|**, **^**)
- bitweise Operatoren (**<<**, **>>**, **>>>**, **&**, **|**, **^**, **~**)
- Zuweisungsoperatoren (**+=**, **-=**, ***=**, **/=**, **%=**, **&=**, **|=**, **^=**, **<<=**, **>>=**, **>>>=**)
- sonstige (**? :**, **[ ]**, **.**, **( )**, **new**, **instanceof**)

Für eine Liste dieser Operatoren sei beispielsweise auf
https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html verwiesen.

Ein **Ausdruck** ist eine Folge von Variablen, Operatoren und Methodenaufrufen, die zur Laufzeit ausgewertet wird und deren Ergebnis einer einfachen Variablen zugewiesen wird.

```
x = (i-- < (a + b) / 2) ? Math.cos(3 * (a - b)) : x + b * b / 10;
```

Die Reihenfolge der Auswertung eines Ausdrucks ist durch gewisse Operator-Vorrangregeln bestimmt. Durch Klammerung kann sie beeinflusst werden. Siehe auch
https://docs.oracle.com/javase/tutorial/java/nutsandbolts/expressions.html.

Grundlagen/Operatoren/OpBsp.java

```
10    int i = 4;
11    int j, k;
12
13    // unäre Operatoren:
14    // präfix ++i
15    // postfix i++
17    j = i++; // i = 5, j = 4
20    j = ++i; // i = 6, j = 6
23
24    // binäre Operatoren:
25    i = 37;
26    j = 3;
27    k = i % 3 + 2 * j;
```

```
29
30      // Der ternäre Operator:
31      int signum;
32      double x;
33      x = 3.0;
34      signum = (x > 0.0) ? 1 : -1;
39
40      // Vergleichsoperatoren:
41      i = 3;
42      k = 6;
44      if (i > k)
45        System.out.println("Es␣gilt:␣i␣>␣k");
48
49      // Zuweisungsoperatoren:
50      i = k;
52      i = i + 2;
54      i += 3;
56      i /= 2;
58
59      // Logische Operatoren mit und ohne Kurzschlussauswertung:
62      x = ((k != 0) && (1 / k > 0.2)) ? k : 0; // keine Division durch 0
63      x = ((k > i) & (j++ > 5)) ? 1 : 0;       // j wird immer erhöht
```


3.7 Mathematische Funktionen, statische Importe

Die üblichen mathematischen Funktionen sind in den Klassen **Math** und **StrictMath** als statische Methoden enthalten. Bei den Methoden von **StrictMath** wird garantiert, dass auf allen Plattformen identische Ergebnisse berechnet werden.

Grundlagen/Math/MathBsp1.java

```
10      double ms2 = Math.sqrt(2.0);
11      double sms2 = StrictMath.sqrt(2.0);
```

Auf die Qualifikation durch den Klassennamen kann verzichtet werden, wenn die Methode **statisch importiert** wird.

Grundlagen/Math/MathBsp2.java

```
7   import static java.lang.Math.sqrt;
13      double ms2 = sqrt(2.0);
```

Die folgende Zeile bewirkt einen statischen Import aller Methoden der Klasse **Math**:

Grundlagen/Math/MathBsp3.java

```
7   import static java.lang.Math.*;
```


3.8 Kontrollstrukturen

Diese Strukturen steuern den Ablauf eines Programmes. In Java gibt es diese Kontrollstrukturen:

- **if** — **else**
- **switch** — **case** — **break**
- Schleifen: **for**, **while**, **do** — **while**
- Ausnahmebehandlung: **try** — **catch** — **finally**, **throw**
- sonstiges: **break**, **continue**, **return**

if — else

Grundlagen/Kontrollstrukturen/IfElseBsp.java

```
12      if (x >= 0)
13        {
14          y = Math.sqrt(x);
15        }
28      if (x == 0)
29        {
30          signum = 0;
31        }
32      else if (x > 0)
33        {
34          signum = 1;
35        }
36      else
37        {
38          signum = -1;
39        }
```

switch — case — break

Grundlagen/Kontrollstrukturen/SwitchBsp.java

```
11    switch (farbe)
12      {
13        case 0:
14          System.out.println("rot");
15          break;
16        case 1:
17          System.out.println("blau");
18          break;
25        default:
26          System.out.println("Kenne diese Farbe nicht");
27      }
```

Seit Java 7 sind als Argument neben ganzen Zahlen, **enum**s und **char**s auch Strings erlaubt.

for-Schleife

Grundlagen/Kontrollstrukturen/ForBsp.java

```
11    int summe = 0;
12    int n = 5;
13    for (int i = 1; i <= n; i++)
14      {
16        summe += i * i;
17      }
21    for (int i = 1, j = 9999; i < j; i *= 2, j -= 444)
```


Erweiterte for-Schleife

Bei der erweiterten Form der **for**-Schleife wird keine Laufvariable mehr benötigt. Das Durchlaufen eines Feldes ist nun mit **for (Typ Bezeichner : Feld) …** möglich.

Grundlagen/Kontrollstrukturen/ForEachBsp.java

```
11    double[] arr = { 1.0, 3.5, 10.5, 0.5, 1.5, 4.0 };
12    double sum = 0;
13
14    // bis Java 1.4:
15    // for (int i = 0; i < arr.length; i++)
16    //   sum += arr[i];
17    for (double el : arr)
18      sum += el;
19
20    double avg = sum / arr.length;
```

Anmerkung: Mit der erweiterten Form der **for**-Schleife können nicht nur Felder durchlaufen werden, sondern als Typ können rechts vom Doppelpunkt alle Klassen angegeben werden, die das Interface **Iterable** implementieren.

while-Schleife

Grundlagen/Kontrollstrukturen/WhileBsp.java

```
11    while (n != 1)
12      {
13        n = n % 2 == 1 ? 3 * n + 1 : n / 2;
14        System.out.println(" n = " + n);
15      }
```


do-while-Schleife

Grundlagen/Kontrollstrukturen/DoWhileBsp.java

```
11    do
12      {
13        System.out.println(n);
14        n--;
15      }
16    while (n >= 0);
```

Ausnahmebehandlungen betrachten wir später (siehe 6).

Mit **break** kann man Schleifen abbrechen, **continue** beendet den aktuellen Schleifendurchlauf.

Switch-Expression

So wie es zu **if** und **else** mit **?:** einen vergleichbaren Ausdruck gibt, gibt es seit Java 14 zu **switch** den **switch-Ausdruck** (oder **switch-Expression**).

Variante 1: mit **yield**

Grundlagen/Kontrollstrukturen/SwitchExprBsp.java

```
12    String text, ergebnis;
13    text = switch (farbe)
14      {
15        case 0: yield "rot";
16        case 1:
17        ergebnis = "blau";
18        yield ergebnis;
21        default: yield "Kenne diese Farbe nicht";
22      };
```

- Jeder Block wird mit **yield** beendet, ein **break** gibt es hier nicht.
- Mehrere **case**-Label können zusammengefasst werden: **case 7, 9, 42:**

Variante 2: mit Pfeilnotation

```
26      text = switch (farbe)
27        {
28          case 0 -> "rot";
29          case 1 ->
30          {
31            ergebnis = "blau";
32            yield ergebnis;
33          }
36          default -> "Kenne diese Farbe nicht";
37        };
```

- angelehnt an Lambda-Ausdrücke (vgl. Kapitel 9)
- kürzer für einzeilige Ergebnisse, daher meist zu bevorzugen
- bei mehreren Zeilen Blockklammern und doch **yield**

Pattern Matching für switch

Seit Java 21 können in **switch**-Ausdrücken und -Kontrollstrukturen Typen unterschieden werden. Mit Hilfe von **when** kann dabei weiter differenziert werden.

Grundlagen/Kontrollstrukturen/PMSwitchBsp.java

```
15      Object o;
29      String info = switch (o)
30        {
31          case null -> "null";
32          case double[] d -> "ein double-Array mit Anfang " + d[0];
33          case int[] _ -> "ein int-Array";
34          case String s when s.length() < 10 -> "der kurze String " + s;
35          case String s -> "der String " + s;
36          default -> "irgendetwas anderes";
37        };
```

- Der **default**-Fall darf hierbei nicht entfallen.
- Im Fall **int[]** interessiert uns nur der Typ und nicht der Wert, daher verwenden wir den Unterstrich als unbenannte Variable (erlaubt seit Java 22).

3.9 Kommandozeilenparameter

Auf Kommandozeilenparameter kann über den **String**-Array-Parameter der Methode **main** (üblicherweise **argv** oder **args**) zugegriffen werden.

Grundlagen/Kommandozeilenparameter/ArgvBsp.java

```
8   public static void main (String[] argv)
9   {
10    for (String arg : argv)
11      {
12        System.out.println(arg);
13      }
27  } // Ende main()
```


Beispiel: Berechnung der Summe der Kommandozeilenparameter vom Typ **int**

Aufruf des Programms z. B. mit *java Summe 4 7 34 -8*

Grundlagen/Kommandozeilenparameter/Summe.java

```
9   public class Summe
10  {
11    public static void main(String[] argv)
12    {
13      int sum = 0;
14
15      if (argv.length == 0)
16        System.out.println("Keine Argumente übergeben");
17      else
18        {
19          for (String arg : argv)
20            sum += Integer.parseInt(arg);
21          System.out.println("Summe = " + sum);
22        }
23    }
24  }
```

3.10 Zusammenfassende Fragen

- Was sind einfache Datentypen?
- Welche einfachen Datentypen gibt es in Java?
- Was ist der Lebensraum einer Variablen?

- Wie deklariert und initialisiert man Variablen?
- Wie kann man Variablen mit unveränderbarem Wert erzeugen?
- Was ist der Unterschied zwischen einfachen und Referenzdatentypen?
- Wie deklariert und initialisiert man Arrays?
- Wie kann man auf Arrays zugreifen?
- Wie deklariert und initialisiert man Strings?
- Wie kann man auf Strings zugreifen?
- Was ist der Unterschied zwischen Identität und Äquivalenz?
- Was bedeutet erweiternde und einschränkende Typkonvertierung?
- Was für Operatoren sind in Java zulässig?
- Welche Operator-Vorrangregeln sind in Java gültig?
- Welche Kontrollstrukturen gibt es?
- Welche Kommentarzeichen gibt es?

4 Objektorientierung in Java, Teil 1

Inhalt

Inhalt/Ziele: Die zentralen und **für das praktische Arbeiten** mit Java wichtigen Begriffe des objektorientierten Softwareentwurfs werden vorgestellt. Das Zusammenspiel zwischen der Deklaration von Klassen einerseits und dem Lebenszyklus von Objekten andererseits steht dabei im Mittelpunkt.

Beim objektorientierten Softwareentwurf versucht man, die in der realen Welt vorkommenden Gegenstände als Objekte zu interpretieren.

Bei der Abbildung der realen Welt nach Software beschränkt man sich dabei auf das Wesentliche:

Beispiel 4.1: Mit einem **Auto** kann man **fahren** und es hat einen **Tachostand**.

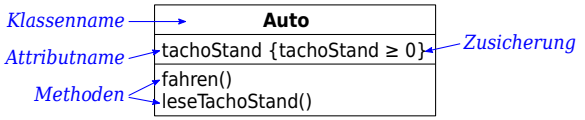
Der Entwickler entwirft nun eine Vorlage, in der alle geforderten Merkmale berücksichtigt sind.

Nach dieser Vorlage können dann beliebig viele konkrete Exemplare neu erzeugt werden.

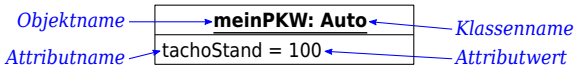
- In einer objektorientierten Programmiersprache heißen solche Vorlagen **Klassen**.
- Die konkreten Exemplare nennen wir **Objekte**.

- Das Erzeugen von Objekten nach der Vorlage einer Klasse heißt **Instantiieren**.
- Die Merkmale können wir unterteilen in **Attribute**, **Methoden** und **Zusicherungen**.

Eine Klasse kann man wie folgt (in **UML**-Notation) darstellen:



Objekte – also die Exemplare der Klassenvorlagen – stellt man ähnlich dar. Zur Unterscheidung wird der Objektname unterstrichen.



Erstes wichtiges Konzept

Im objektorientierten Softwareentwurf werden in den Klassen bzw. Objekten **sowohl** Attribute **als auch** Methoden **und** Zusicherungen zusammengefasst.

Weitere Beispiele für Klassen und deren Attribute

Ausgangspunkt für die Modellierung ist in der Regel die Spezifikation (Beschreibung) des Problems, z. B. hier für die Software einer Bibliothek. Dabei werden sich verschiedene Arten von Klassen finden:

Typ	Klasse	Attribute
Dinge aus der realen Welt	Buch	Titel, Autor, Erscheinungsjahr, Signatur, aktueller Ausleiher
Rollen von Personen	Ausleiher	Name, Nummer, Liste der geliehenen Bücher
	Mitarbeiter	Name, Abteilung
Organisationseinheiten	Abteilung	Name, Liste der Mitarbeiter
Orte	Büro	Gebäude, Ebene, Nummer
Geräte	Drucker	Netzwerkadresse, Typ
Ereignisse	Fristüberschreitung	Buch, Ablaufdatum

4.1 Klassen und Objekte in Java

Softwareentwicklung unter Java erfolgt also auf zwei Ebenen:

Zum einen entwerfen und implementieren wir Klassen. Ein Java-Quellcode für unsere Klasse **Auto** könnte so aussehen:

Objektorientierung/Autos/01/Auto.java

```
6  public class Auto
7  {
8    // Objektattribute
9    double tachoStand;
10
11   // Objektmethoden
12   public void fahren(double distanz)
13   {
14     tachoStand = tachoStand + distanz;
15   }
16
17   public double leseTachoStand()
18   {
19     return tachoStand;
20   }
21
22   public void ausgeben()
23   {
24     System.out.println("Tachostand = " + tachoStand + " km");
25   }
38 } // Ende: public class Auto
```


Auf der anderen Ebene können wir von einer Klasse beliebig viele Objekte instantiieren und auf ihre Merkmale zugreifen:

```
28   public static void main(String[] argv)
29   {
30     Auto meinPKW = new Auto();
31     meinPKW.fahren(100);
32     meinPKW.ausgeben();
33     meinPKW.fahren(20);
34     System.out.println(meinPKW.leseTachoStand());
35     Auto bmw = new Auto();
36     bmw.ausgeben();
37   }
```

4.1.1 Deklaration von Klassen

Klassen implementieren wir in Java-Quelltextdateien ***.java**. In einer Datei **Xyz.java** muss mindestens eine Klasse mit dem Namen **Xyz** deklariert werden. Üblicherweise verwendet man für jede Klassendeklaration eine eigene Datei.

Die Klassendeklaration wird durch das Schlüsselwort **class** angezeigt. Für Eigenschaften, die sich auf Zugriffsrechte oder Vererbung beziehen, können zusätzliche Modifizierer wie **public** oder **final** verwendet werden.

Nach der Festlegung des Namens erfolgt in geschweiften Klammern die Implementierung des Klassenkörpers.

Im Beispiel **Auto** sind bereits einige **Objektattribute** und **Objektmethoden** implementiert.

Um eine Klasse lokal auszutesten, kann man die Methode **main()** benutzen.

Java bietet dem Entwickler die Möglichkeit, parallel zum Implementieren **Dokumentationskommentare** zu verfassen, die mit dem JDK-Werkzeug **javadoc** ausgewertet werden können.

Objektorientierung/Autos/02-javadoc/Auto.java

```
1  /**
2   * Klasse Auto (2).
3   * Eine Basis-Implementierung einer Klasse zur Simulation
4   * eines Autos.
5   * <p>
6   * Mit Hilfe von Dokumentationskommentaren kann der Entwickler
7   * einer Klasse dem Anwender den Leistungsumfang der Klasse und
8   * die Benutzung ihrer Schnittstellen darstellen.
9   *
10  * @author Benedikt Großer, Holger Arndt
11  * @version 10.04.2024
12  */
13 public class Auto
19   /** simuliert die Fortbewegung um eine gewisse Distanz.
20    *  @param distanz die zu fahrende Distanz.
21    */
22   public void fahren(double distanz)
27   /** erlaubt Zugriff auf den aktuellen Tachostand des Autos.
28    *  @return den aktuellen Tachostand des Autos.
29    */
30   public double leseTachoStand()
35   /** gibt alle Informationen des Autos aus.
36    */
37   public void ausgeben()
```

Quelltextanalyse

- Mit dem Kommando **`javadoc Auto.java`** wird ein Dokumentationssystem aus HTML-Seiten generiert:

  `Quellen/Objektorientierung/Autos/02-javadoc/javadoc/index.html`

- Innerhalb der Dokumentationskommentare im Java-Quellcode dürfen die üblichen HTML-Tags verwendet werden.
- Zur Strukturierung gibt es Markierungen **(Tags)**, die mit dem Zeichen **@** eingeleitet werden:

Tag	Dokumentation	Verwendung bei
`@author`	Autoreneintrag	Klasse, Interface
`@version`	Versionseintrag (Datum)	Klasse, Interface
`@see`	Querverweis	Klasse, Interface, Attribut, Methode
`@param`	Parameter der Methode	Methode
`@return`	Rückgabewert der Methode	Methode
`@throws` (alt: `@exception`)	Ausnahmen	Methode
`@deprecated`	zeigt eine veraltete Methode an	Methode

- Details zur Benutzung von **`javadoc`** findet man beispielsweise unter `https://docs.oracle.com/en/java/javase/22/javadoc/`.

4.1.2 Instantiieren von Objekten

Klassen dienen als Vorlage für viele einzelne Exemplare (Objekte). Objekte sind Referenzdatentypen.

Objektorientierung/Autos/03-Instantiierung/Auto.java

```
30    Auto bmw; // erzeugt eine Referenz auf ein Auto, nicht das Objekt selbst
```

Instantiierung von Objekten erfolgt mit dem **`new`**-Operator.

```
32    bmw = new Auto(); // instantiiert das Objekt
33
34    Auto zweiCV = new Auto(); // beide Schritte kombiniert
```

Zugriff auf Attribute und Methoden erfolgt mit Hilfe der Punkt-Notation.

Objektattribute lassen sich via **`Objekt.Attribut`** und Objektmethoden via **`Objekt.Methode()`** ansprechen.

```
36    double tsbmw = bmw.tachoStand;
37    zweiCV.tachoStand = 44444;
38
39    tsbmw = bmw.leseTachoStand();
40    zweiCV.fahren(500);
```

Zweites wichtiges Konzept:

Gutes objektorientiertes Design zeichnet sich durch **Datenkapselung** aus. Der Zugriff auf Attribute sollte ausschließlich über Methoden erfolgen, um z. B. die zugesicherten Merkmale eines Objektes zu garantieren. Mit Hilfe von **Zugriffsrechten** (siehe 5.3) kann man dies erreichen.

4.2 Objekte sind Referenzdatentypen

Objekte werden „per Referenz“ verarbeitet, d. h. die Adresse des Objekts wird in einer Variablen abgelegt. Standardwert ist **`null`**. Erst zur Laufzeit durch Anwenden des **`new`**-Operators wird Speicherplatz für eine Instanz angefordert.

Wieder muss man zwischen Identität und Äquivalenz unterscheiden.

Objektorientierung/Autos/04-Referenzen/Auto.java

```
31    Auto meinPKW = new Auto();
32    Auto bmw = new Auto();
33    Auto zweiCV;
38    System.out.println(bmw == meinPKW); // false
39    zweiCV = bmw;
40    System.out.println(bmw == zweiCV);  // true
41    zweiCV = null;
42    System.out.println(zweiCV == null); // true
```

- **Identität** zweier Variablen vom Referenzdatentyp ist gegeben, wenn sie auf dasselbe Objekt verweisen.
- **Äquivalenz** prüft, ob zwei verschiedene Objekte den gleichen Inhalt haben.

Wir werden später (siehe 5.9) sehen, wie man dazu die Methode **`equals()`** verwenden kann.

4.3 Über Methoden

Die Deklaration einer **Methode** enthält mindestens:

- **Typ**
- **Name**
- **Parameterliste** (evtl. leer)
- **Methodenkörper**

In der **Methodensignatur** sind optional zu verwenden:

- **Modifizierer** für Zugriffsrechte usw. (siehe auch 5.3)
- **Klauseln** zur Ausnahmeerzeugung usw. (siehe auch 6.4)