Informatik

Java-Überblick und *UML*-Überblick

Klett

1 Datentypen

In *Java* gibt es zwei Arten von Typen: primitive Datentypen und Referenz- bzw. Objekttypen.

Primitive Datentypen

Typname	Beschreibung	Länge in Byte	Wertebereich
`boolean`	Boole'scher Wert (wahr oder falsch, 1 Bit)	1	`true, false`
`char`	einzelnes Zeichen (16 Bit)	2	alle Unicode-Zeichen, z. B. `'a', ..., 'z', 'A', ..., 'Z', '3', '%' ...` (Zeichen werden in einfache Hochkommata gesetzt)
`byte`	ganze Zahl (8 Bit)	1	$-2^7, \ldots, 2^7 - 1$ (−128, …, 127)
`short`	ganze Zahl (16 Bit)	2	$-2^{15}, \ldots, 2^{15} - 1$ (−32 768, …, 32 767)
`int`	ganze Zahl (32 Bit)	4	$-2^{31}, \ldots, 2^{31} - 1$ (−2 147 483 648, …, 2 147 483 647)
`long`	ganze Zahl (64 Bit)	8	$-2^{63}, \ldots, 2^{63} - 1$ (−9 223 372 036 854 775 808, …, 9 223 372 036 854 775 807)
`float`	Fließkommazahl (32 Bit)	4	$-3{,}4028 \cdot 10^{38}, \ldots, 3{,}4028 \cdot 10^{38}$; $-45 < \text{Exponent} \leqq 38$
`double`	Fließkommazahl (64 Bit)	8	$-1{,}7977 \cdot 10^{308}, \ldots, 1{,}7977 \cdot 10^{308}$; $-324 < \text{Exponent} \leqq 308$

Alle primitiven Datentypen haben entsprechend dem Speicherplatzbedarf eine feste Länge.

1 Bit ist eine Informationseinheit (wahr oder falsch, 0 oder 1). 8 Bit ergeben 1 Byte. Wird eine Variable beispielsweise als `byte` deklariert, so wird für diese 1 Byte Speicher zur Verfügung gestellt. Folglich kann man hier nur Zahlen verwenden, für die im Dualsystem maximal sieben Stellen benötigt werden, das achte Bit wird für das Vorzeichen benötigt.

Die möglichen Operatoren sind auf Seite 162 aufgelistet.

Referenztypen (Objekttypen)
Zu den Referenztypen gehören Objekte, Strings und Arrays.

Typname	Beschreibung	Beispiel
`String`	Zeichenkette (Text)	`"Hallo!"`
Array (z.B. `int[]`)	Feld (hier ganzzahlig)	`{1, 2, 3, 4, 5}`

Strings und Arrays sind streng genommen auch Objekte, können aber ohne Aufruf des `new`-Operators erzeugt werden.

Autoren: Peter Hubwieser, Patrick Löffler, Petra Schwaiger, Matthias Spohrer, Markus Steinert, Siglinde Voß, Ferdinand Winhard
Gestaltung: Andrea Eckhardt
Illustration: Andrea Eckhardt

Seite aus: Informatik 5 SB
ISBN: 978-3-12-731068-9

Variablendeklaration

(1) Bei primitiven Datentypen

Syntax:

```
(<Zugriffsmodifikator>) <Datentyp> <Bezeichner> ( = <Wert>)
```

`<Zugriffsmodifikator>`	`private` oder `public` (oder `protected`) Innerhalb der Klassendefinition (als globale Variable) prinzipiell `private`, innerhalb einer Methode als lokale Variable wird der Zugriffsmodifikator meist weggelassen.
`<Datentyp>`	Vergleichen Sie die Liste auf Seite 160, z. B. `int`, `boolean` oder `char`
`<Bezeichner>`	Beliebiger Name, sollte aber den Inhalt der Variablen charakterisieren. So sagt etwa *jahre* als Bezeichner mehr aus als *x*.
`<Wert>`	Der Variablen kann gleich ein Wert zugewiesen werden, die Zuweisung ist jedoch nicht zwingend notwendig.

Beispiel	**Erläuterung**
`private int summe;`	Es wird eine ganzzahlige Variable mit dem Bezeichner *summe* deklariert.
`private int jahre = 13;`	Es wird eine ganzzahlige Variable mit dem Bezeichner *jahre* deklariert und ihr der Wert *13* zugewiesen.
`private boolean weiblich;`	Es wird eine Boole'sche Variable mit dem Bezeichner *weiblich* deklariert.
`private boolean neu = true;`	Es wird eine Boole'sche Variable mit dem Bezeichner *neu* deklariert, die den Wert `true` erhält.
`char b = 'x';`	Eine Variable vom Typ *Zeichen* wird deklariert und mit dem Wert *x* belegt.
`double pi = 3.14159;`	Eine reellwertige Variable namens *pi* wird deklariert und erhält den Wert *3,14159*.

(2) Bei Referenztypen

Zur Erzeugung eines Objektes wird im Allgemeinen der `new`-Operator verwendet.
Zeichenketten (Strings) können oft wie primitive Datentypen behandelt werden, manchmal aber auch nicht (z. B. hinsichtlich des Vergleichs mit `==`).

Beispiel	**Erläuterung**
`String vorname;`	Es wird eine String-Variable mit dem Bezeichner *vorname* deklariert.
`String nachname = "Müller";`	Eine Variable vom Typ *Zeichenkette* wird deklariert und mit dem Wert *Müller* belegt.
`int[] primz = {2, 3, 5, 7, 11, 13};`	Es wird ein ganzzahliges Feld der Länge 6 mit dem Bezeichner *primz* erzeugt und das Feld mit den ersten sechs Primzahlen aufgefüllt.
`int[] noten = new int[10];`	Es wird ein leeres Feld mit dem Bezeichner *noten* erzeugt, welches insgesamt zehn ganzzahlige Werte aufnehmen kann.
`double[] messwerte;`	Es wird ein Feld mit dem Namen *messwerte* deklariert. Bevor es jedoch Werte aufnehmen kann, muss es mit dem `new`-Operator erzeugt werden.
`int[][] = new int[2][3]`	Es wird ein zweidimensionales Feld der Größe $2 \cdot 3$, d. h. eine Matrix mit zwei Zeilen und drei Spalten erzeugt, welche ganzzahlige Werte aufnehmen kann.
`Kreis kreis1;`	Es wird eine Variable mit dem Bezeichner *kreis1* vom Typ *Kreis* deklariert. Mit `kreis1 = new Kreis()` wird der Konstruktor aufgerufen, das Objekt erzeugt und dem Bezeichner *kreis1* zugewiesen.

Autoren: Peter Hubwieser, Patrick Löffler, Petra Schwaiger, Matthias Spohrer, Markus Steinert, Siglinde Voß, Ferdinand Winhard
Gestaltung: Andrea Eckhardt
Illustration: Andrea Eckhardt

Seite aus: Informatik 5 SB
ISBN: 978-3-12-731068-9

2 Operatoren und Typumwandlung

Operatoren

Operator	Erläuterung	Bemerkung/Beispiel
`+`	positives Vorzeichen	`+i` ist gleichbedeutend mit `i`.
`-`	negatives Vorzeichen	`-i` dreht das Vorzeichen von `i` um.
`+`	Addition	`x + y` ergibt die Summe von x und y.
`-`	Subtraktion	`x - y` ergibt die Differenz von x und y.
`*`	Multiplikation	`x * y` ergibt das Produkt von x und y.
`/`	Division	`x / y` ergibt den Quotienten von x und y. Achtung: Sind x und y beide ganzzahlig, so ist auch x / y ganzzahlig, d.h., `x / y` liefert die ganzzahlige Division ohne Rest (Beispiel: `7 / 3` liefert 2). Ist x oder y jedoch ein Fließkommawert, so ist auch x / y ein Fließkommawert (Beispiel: `7.0 / 3` liefert 2.33333333333333333).
`%`	modulo	`x % y` ergibt den ganzzahligen Rest bei Division von x durch y. (`7 % 3` liefert 1.)
`++`	Inkrement	`i++` entspricht `i = i + 1` und erhöht den Wert von i um 1.
`--`	Dekrement	`i--` entspricht `i = i - 1` und erniedrigt den Wert von i um 1.
`=`	Zuweisung	`x = y` weist x den Wert von y zu.
`==`	Vergleich	`x = = y` ergibt wahr, wenn x gleich y ist bzw. wenn bei Referenztypen beide Werte auf dasselbe Objekt zeigen.
`<`	kleiner	`x < y` ergibt wahr, wenn x kleiner ist als y.
`<=`	kleiner gleich	`x <= y` ergibt wahr, wenn x kleiner oder gleich y ist.
`>`	größer	`x > y` ergibt wahr, wenn x größer ist als y.
`>=`	größer gleich	`x >= y` ergibt wahr, wenn x größer oder gleich y ist.
`!=` `<>`	ungleich	`x != y` (oder `x<>y`) ergibt wahr, wenn x ungleich y ist bzw. wenn bei Referenztypen beide Werte auf verschiedene Objekte zeigen.
`!`	logisches NICHT	`!x` ergibt wahr, wenn x falsch ist und umgekehrt.
`&&`	logisches UND	`x && y` ergibt wahr, wenn sowohl x als auch y wahr sind.
`||`	logisches ODER	`x || y` ergibt wahr, wenn mindestens einer der beiden Ausdrücke x oder y wahr ist.
`^`	exklusives ODER	`x^y` ergibt wahr, wenn x wahr ist und zugleich y falsch oder umgekehrt.
`new`	`new`-Operator	zur Erzeugung von Objekten
`instanceof`	`instanceof`-Operator	`x instanceof y` liefert wahr, wenn x eine Instanz der Klasse y oder einer ihrer Unterklassen ist. So lässt sich herausfinden, zu welcher Klasse ein bestimmtes Objekt gehört.

Autoren: Peter Hubwieser, Patrick Löffler, Petra Schwaiger, Matthias Spohrer, Markus Steinert, Siglinde Voß, Ferdinand Winhard
Gestaltung: Andrea Eckhardt
Illustration: Andrea Eckhardt

Seite aus: Informatik 5 SB
ISBN: 978-3-12-731068-9

Typumwandlung (Casting)
Automatische (implizite) Typumwandlung erfolgt beispielsweise, wenn eine `short`-Variable und eine `int`-Variable gemeinsam in einem Additionsausdruck verwendet werden. Der kleinere Datentyp (`short`) wird dabei dem größeren (`int`) angepasst. Diese Konvertierung erfolgt automatisch durch den Compiler.

Explizite Typumwandlung erfolgt meist durch Verlust von Informationen und wird nicht vom Compiler selbstständig vorgenommen. Man verwendet dazu den **Type-Cast-Operator**.

Beispiel	Erläuterung

`double x = 7.0;`
`double y = 3.0;`
`int z;`
`z = (int)(x + y);`
	Zwei Fließkommazahlen x und y sowie eine ganze Zahl z werden deklariert.
	Die Zuweisung `z = x + y` würde ohne Typumwandlung Probleme bereiten, daher wird der Type-Cast-Operator (`int`) verwendet, der das Ergebnis von `x + y` in eine ganze Zahl konvertiert.

`(int)(7.0 / 3.0);`	Dieser Term wird zu 2 ausgewertet.
`int b = 65;` `System.out.println((char)(b + 1));`	b + 1 ergibt 66. Anschließend wird wegen des Operators (`char`) das Zeichen mit der Nummer 66 auf dem Bildschirm ausgegeben, nämlich das Zeichen B.

3 Methodendefinition

Syntax:

```
(<Zugriffsmodifikator>) <Rückgabetyp> <Bezeichner> (<Parameter>) {...}
```

Beispiel	Erläuterung

`public void hello(String name) {`
`  System.out.print("Hallo " + name);`
`}`
	Die öffentliche Methode *hello* gibt auf dem Bildschirm „Hallo XYZ“ aus, wenn ihr „XYZ“ beim Aufruf übergeben wurde.

`public double mittelwert(double x,`
`  double y) {return (x + y) / 2;`
`}`
	Die Methode *mittelwert* gibt den Mittelwert zweier reeller Zahlen x und y zurück.

`public void neuzeichnen() {`
`  loeschen();`
`  zeichnen();`
`}`
	Die Methode *neuzeichnen* hat keinen Rückgabewert, keine Parameter und ruft nacheinander die Methoden *löschen* und *zeichnen* auf.

Eine besondere *Java*-Methode ist die `main`-Methode. Wird in einer Klasse eine `main`-Methode definiert, dann wird diese Methode beim Aufruf des *Java*-Interpreters mit dem zugehörigen Klassennamen als erstes ausgeführt, d. h. die Klasse bzw. das Programm „gestartet“. Soll ein direkt ausführbares und lauffähiges Programm erzeugt werden, so muss eine solche Hauptklasse mit `main`-Methode existieren.

Autoren: Peter Hubwieser, Patrick Löffler, Petra Schwaiger, Matthias Spohrer, Markus Steinert, Siglinde Voß, Ferdinand Winhard
Gestaltung: Andrea Eckhardt
Illustration: Andrea Eckhardt

Seite aus: Informatik 5 SB
ISBN: 978-3-12-731068-9

Syntax:

```
public static void main(String args[]) {
   // Hier beginnt unser Programm

}
```

`static` bedeutet hierbei, dass zum Aufruf der Methode kein Objekt dieser Klasse existieren muss.

4 Klassendefinition

Normale Klassendefinition

Syntax:

```
(<Zugriffsmodifikator>) class <Bezeichner> {
   // Attribute
   // Methoden
}
```

Beispiel	**Erläuterung**
`public class Kugel {`	Kopf
`  //Attribute`	Deklaration der Attribute:

`  private int radius;`
`  private double xPos;`
`  private double yPos;`
	Es werden ein ganzzahliges Attribut für den Radius sowie jeweils eine reellwertige Variable für die x- bzw. y-Position der Kugel deklariert.

`  //Methoden`	Methodendefinition

`  Kugel(int kugelradius) {`
`    radius = kugelradius ;`
`    xPos = 0.0;`
`    yPos = 0.0;`
`  }`
	Der Konstruktor zur Erzeugung des Objektes hat den gleichen Namen wie die Klasse selbst.

`  public double rauminhalt() {`
`  return 4 * radius * radius * radius *`
`    Math.PI / 3;`
`  }`
	Methode, die als Rückgabewert den Rauminhalt der Kugel berechnet

`} // Ende Kugel`	Ende der Klassendefinition

Definition einer Unterklasse bei Vererbung:

Syntax:

```
(<Zugriffsmodifikator>) class <Unterklasse> extends <Oberklasse> {
   // Attribute
   // Methoden
}
```


Autoren: Peter Hubwieser, Patrick Löffler, Petra Schwaiger, Matthias Spohrer, Markus Steinert, Siglinde Voß, Ferdinand Winhard
Gestaltung: Andrea Eckhardt
Illustration: Andrea Eckhardt

Seite aus: Informatik 5 SB
ISBN: 978-3-12-731068-9

Beispiel

```
class Ball extends Kugel {
   // Attribute
   private String farbe;
   private String material;

   // Methoden
   Ball(int rad, String far, String mat) {
      super(rad);
      farbe = far;
      material = mat;
   }
   public void farbeSetzen(String f_neu) {
      farbe = f_neu;
   }
} // Ende Ball
```

Erläuterung

Die neue Klasse BALL wird von KUGEL (Seite 164) abgeleitet.
Deklaration der Attribute:
Die Klasse BALL hat dieselben Attribute wie die Klasse KUGEL und zusätzlich noch ein Attribut für die Farbe und das Material.
Methodendefinition
Konstruktor
`super` ruft den Konstruktor der Oberklasse mit den angegebenen Parameterwerten auf, hier also `Kugel(rad)`. Den von der Oberklasse geerbten Attributen werden also folgende Werte zugewiesen: `radius = rad; xPos = 0.0; yPos = 0.0.`
Methode, die die Farbe des Balles ändert

Ende der Klassendefinition

Abstrakte Klassen

Abstrakte Klassen werden durch das Schlüsselwort `abstract` definiert. Von derartigen Klassen können keine Instanzen erzeugt werden.

Syntax:

```
(<Zugriffsmodifikator>) abstract class <Klassenbezeichner> {
   // Attribute
   // Methoden
}
```

Abstrakte Klassen können abstrakte Methoden besitzen, die ebenso mit dem Schlüsselwort `abstract` gekennzeichnet und in der abstrakten Klasse selbst nicht implementiert werden:

Beispiel

```
public abstract class Mitarbeiter {
   //Attribute
   protected double grundgehalt;
   protected double verdienst;

   // Methoden
   public abstract double verdienstBerechnen();

} // Ende Mitarbeiter
```

Erläuterung

Kopf
Deklaration der Attribute:
Die Attribute *grundgehalt* und *verdienst* sind geschützte Attribute (`protected`), auf die (nur) von allen Unterklassen aus zugegriffen werden kann.
Methodendefinition
abstrakte Methode, die in den Unterklassen implementiert wird
Ende der Klassendefinition

*** Interfaces**

Interfaces sind Klassen, die neben Konstantendefinitionen ausschließlich Methodensignaturen enthalten.
Sie legen ein allgemeines Verhalten fest, d.h. welche Methoden vorhanden sein sollen und was sie bewirken, ohne diese jedoch inhaltlich genau zu definieren. Man muss nur ihre Signatur kennen, um sie aufrufen zu können. Wie die Methode im Einzelnen abgearbeitet wird, ist für den Aufrufer irrelevant.

Autoren: Peter Hubwieser, Patrick Löffler, Petra Schwaiger, Matthias Spohrer, Markus Steinert, Siglinde Voß, Ferdinand Winhard
Gestaltung: Andrea Eckhardt
Illustration: Andrea Eckhardt

Seite aus: Informatik 5 SB
ISBN: 978-3-12-731068-9

Syntax:

```
(<Zugriffsmodifikator>) interface <Bezeichner> {
  // abstrakte Methoden
}
```

Klassen erben von Interfaces mithilfe des Schlüsselwortes `implements` (statt `extends`):

Syntax:

```
(<Zugriffsmodifikator>) class <Unterklasse> implements <Interface> {
  // Attribute
  // Methoden
}
```

Beispiel

```
public interface Koerper {
  // Attribute gibt es nicht
  // Methoden
  public double oberflaeche();
  public double volumen();
} // Ende Koerper
```

Erläuterung

Kopf

Methodendefinition
Es werden die Signatur für *oberfläche* und *volumen* festgelegt. Der Rumpf der Methoden wird weggelassen.
Ende der Interface-Definition

```
public class Kugel implements Koerper {
  // Attribute
  private double radius ;
  // Methoden
  public double oberflaeche() {
  return 4 * Math.PI * radius * radius;
  }
  public double volumen() {
  return 4 * Math.PI * radius * radius *
    radius / 3;
  }
} // Ende Kugel
```

Kopf (Alternative zur Klasse KUGEL von Seite 164)

Methodendefinition
Hier werden die im Interface deklarierten Methoden *oberfläche* und *volumen* überschrieben.

Ende der Klassendefinition

```
public class Koerperrechner {


  // Attribute
  // Methoden
  public double oberflBer(Koerper k) {
  return k.oberflaeche();
  }
  public double volumenBer(Koerper k) {
  return k.volumen();
  }
} // Ende Koerperrechner
```

Die Klasse KÖRPERRECHNER verfügt über Methoden, um die Oberfläche und das Volumen beliebiger Körper zu berechnen.

Es wird lediglich die Methode der Schnittstelle *Körper* aufgerufen, die dann durch das konkrete Objekt *k* abgearbeitet wird. Erst zur Laufzeit des Programms ist bekannt, ob es sich bei dem als Parameter übergebenem Körper *k* z. B. um einen Quader, eine Kugel oder einen Zylinder handelt.

Ende der Klassendefinition

Autoren: Peter Hubwieser, Patrick Löffler, Petra Schwaiger, Matthias Spohrer, Markus Steinert, Siglinde Voß, Ferdinand Winhard
Gestaltung: Andrea Eckhardt
Illustration: Andrea Eckhardt

Seite aus: Informatik 5 SB
ISBN: 978-3-12-731068-9

5 Kontrollstrukturen

Sequenz
Jede Anweisung wird mit einem Semikolon abgeschlossen.
Mehrere Anweisungen nacheinander ergeben eine Sequenz.

Beispiel

```
kreis1.farbeSetzen("rot");
kreis2.farbeSetzen("schwarz");
ampel1.zustandSetzen("Rotlicht");
```

Struktogramm

Farbe setzen von kreis1 auf rot
Farbe setzen von kreis2 auf schwarz
Zustand setzen von ampel1 auf rotlicht

Bedingte Anweisung (Fallunterscheidung)
Syntax:
Die bedingte Anweisung gibt es in zwei Formen: mit oder ohne Alternative.

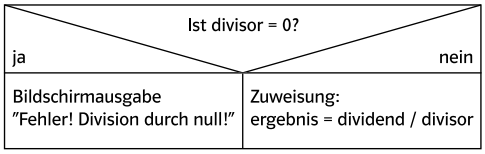
(1) Mit Alternative
Syntax:

```
if (<Bedingung>) {
   <Anweisungen>
}
else {
   <Anweisungen>
}
```

Beispiel

```
if (divisor == 0) {
   System.out.println("Fehler!
      Division durch null!");
}
else {
   ergebnis = dividend / divisor;
}
```

Struktogramm



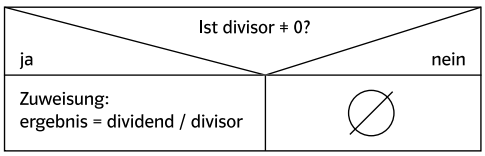
(2) Ohne Alternative
Hier wird der `else`-Teil einfach weggelassen.
Syntax:

```
if (<Bedingung>) {
   <Anweisungen>
}
```

Beispiel

```
if (divisor != 0) {
   ergebnis = dividend / divisor;
}
```

Struktogramm



Autoren: Peter Hubwieser, Patrick Löffler, Petra Schwaiger, Matthias Spohrer, Markus Steinert, Siglinde Voß, Ferdinand Winhard
Gestaltung: Andrea Eckhardt
Illustration: Andrea Eckhardt

Seite aus: Informatik 5 SB
ISBN: 978-3-12-731068-9

Mehrfachauswahl

Die `switch`-Anweisung kann beliebig viele Fälle untersuchen.
Die zu überprüfende Variable muss vom Typ `byte`, `short`, `int` oder `char` sein.

Syntax:

```
switch (<Variable>) {
  case <Wert1>:
    <Anweisungen1>;
    break;
  case <Wert2>:
    <Anweisungen2>;
    break;
  ...
  default:
    <Anweisungen3>;
    break;
}
```

```
switch (<Variable>) {
  case <Wert1>:
  case <Wert2>:
    <Anweisungen1>;
    break;
  case <Wert3>:
  case <Wert4>:
  case <Wert5>:
    <Anweisungen2>;
    break;
  ...
  default:
    <Anweisungen3>;
    break;
}
```

Beispiel

```
switch (wochentag) {
  case 6:
    wochenende = true;
    break;
  case 7:
    wochenende = true;
    break;
  default:
    wochenende = false;
    break;
}
```

```
switch (note) {
  case 1:
    wortlaut = "sehr gut";
    break;
  case 2:
    wortlaut = "gut";
    break;
  case 3:
    wortlaut = "befriedigend";
    break;
  case 4:
    wortlaut = "ausreichend";
    break;
```

Struktogramm und Erläuterung

wochentag = ?		
6	7	default
wochenende = wahr	wochenende = wahr	wochenende = falsch

Falls die ganzzahlige Variable *wochentag* den Wert *6* oder *7* annimmt (für „Samstag“ oder „Sonntag“), wird der Boole'schen Variablen *wochenende* der Wert *wahr*, ansonsten der Wert *falsch* zugewiesen.

note = ?					
1	2	3	4	5	6
wortlaut = "sehr gut"	wortlaut = "gut"	wortlaut = "befriedigend"	wortlaut = "ausreichend"	wortlaut = "mangelhaft"	wortlaut = "ungenügend"

Die ganzzahlige Variable *note* wird untersucht und entsprechend ihrem Wert wird der String-Variablen *wortlaut* die Übersetzung der Note zugewiesen.

Autoren: Peter Hubwieser, Patrick Löffler, Petra Schwaiger, Matthias Spohrer, Markus Steinert, Siglinde Voß, Ferdinand Winhard
Gestaltung: Andrea Eckhardt
Illustration: Andrea Eckhardt

Seite aus: Informatik 5 SB
ISBN: 978-3-12-731068-9

```
  case 5:
    wortlaut = "mangelhaft";
    break;
  case 6:
    wortlaut = "ungenügend";
    break;

}
```

```
switch (buchstabe) {
  case 'a':
  case 'e':
  case 'i':
  case 'o':
  case 'u':
    art = "Vokal";
    break;
  case 'ä':
  case 'ö':
  case 'ü':
    art = "Umlaut";
    break;
  default:
    art = "Konsonant";
    break;
}
```

buchstabe = ?		
'a' – 'e' – 'i' – 'o' – 'u'	'ä' – 'ö' – 'ü'	default
Vokal	Umlaut	Konsonant

Bemerkungen

Die `default`-Klausel deckt alle nicht explizit aufgeführten Fälle ab und kann gegebenenfalls auch weggelassen werden. Eventuell wird dann gar kein Fall ausgeführt.

Die `break`-Anweisung nach einem Fall sorgt dafür, dass weitere Fälle nicht mehr durchlaufen werden. Nach der `default`-Anweisung ist ein `break` nicht notwendig, wird jedoch üblicherweise als guter Programmierstil betrachtet.

Autoren: Peter Hubwieser, Patrick Löffler, Petra Schwaiger, Matthias Spohrer, Markus Steinert, Siglinde Voß, Ferdinand Winhard
Gestaltung: Andrea Eckhardt
Illustration: Andrea Eckhardt

Seite aus: Informatik 5 SB
ISBN: 978-3-12-731068-9

Wiederholung mit fester Anzahl

Syntax:

```
for (<Initialisierung>; <Bedingung>; <Update>) {
  <Anweisungen>
}
```

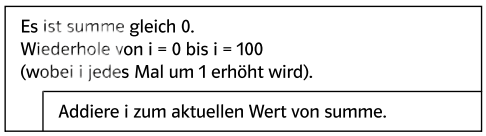
`<Initialisierung>`	Deklaration einer ganzzahligen Lauf- oder Zählvariablen und Zuweisung ihres Anfangswertes (oft `int i = 0`)
`<Bedingung>`	Solange die Bedingung (abhängig von der Zählvariablen) erfüllt ist, werden nachfolgende Anweisungen ausgeführt. Die Bedingung wird jeweils vor dem Durchlauf getestet, daher auch „Abbruchbedingung“ genannt (z. B. `i < 10`)
`<Update>`	Das Update erfolgt nach jedem Durchlauf und ändert die Laufvariable entsprechend der angegebenen Zuweisung (oft `i++`).

Beispiel

```
int summe = 0;

for (int i = 0; i <= 100; i++) {
  summe = summe + i;
}
```

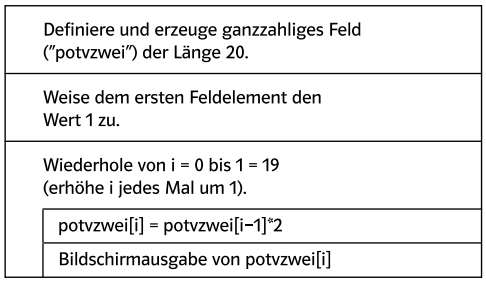
Struktogramm und Erläuterung



Berechnet die Summe aller ganzen Zahlen von 0 bis 100. Als Laufvariable wird die ganze Zahl *i* deklariert, ihr Anfangswert ist 0. Die Anweisung soll so lange wiederholt werden, bis *i* den Wert 100 erreicht (einschließlich), wobei *i* bei jedem Durchlauf um 1 erhöht wird. Der aktuelle Wert von *i* wird dabei dem Wert der Variablen *summe* hinzugezählt (die Anfangsbelegung von *summe* sollte 0 sein).

```
int[] potvzwei = new int[20];
potvzwei[0] = 1;


for (int i = 1; i < 20; i = i + 1) {
  potvzwei[i] = potvzwei[i - 1] * 2;
  System.out.println(potvonzwei[i]);
}
```



Es wird ein ganzzahliges Feld namens *potvzwei* (Potenzen von 2) der Länge 20 erzeugt, das erste Feldelement (mit Index 0) erhält den Wert *1* (2^0).

Beim Durchlauf der `for`-Wiederholung wird jedem der verbleibenden 19 Feldelemente die dem Index entsprechende Potenz von 2 zugewiesen und auf dem Bildschirm ausgegeben.

Autoren: Peter Hubwieser, Patrick Löffler, Petra Schwaiger, Matthias Spohrer, Markus Steinert, Siglinde Voß, Ferdinand Winhard
Gestaltung: Andrea Eckhardt
Illustration: Andrea Eckhardt

Seite aus: Informatik 5 SB
ISBN: 978-3-12-731068-9

Wiederholung mit Anfangsbedingung

Syntax:

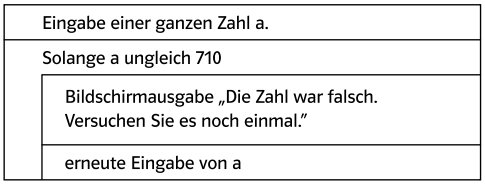
```
while (<Bedingung>) {
   <Anweisungen>
}
```

Die Bedingung wird vor der Ausführung der Anweisungen getestet, sodass nachfolgende Anweisungen möglicherweise gar nicht ausgeführt werden.

Beispiel

```
int a = eingabe();
while (a != 710) {
   System.out.println("Die Zahl
      war falsch. Versuchen Sie
      es noch einmal.");
   a = eingabe();
}
```

Struktogramm und Erläuterung



Zuerst wird eine Variable a vom Typ *Integer* deklariert. Die Zuweisung erfolgt über eine Methode *eingabe()*, welche ermöglicht, eine ganze Zahl über die Tastatur einzugeben und diese als Rückgabewert liefert.

Solange a nicht den Wert *710* hat, wird der angegebene Satz auf dem Bildschirm ausgegeben und zur erneuten Eingabe einer Zahl aufgefordert.

Wiederholung mit Endbedingung

Syntax:

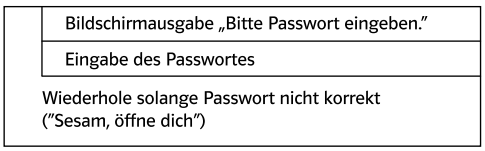
```
do {
   <Anweisungen>
} while (<Bedingung>);
```

Die Bedingung wird nach der ersten Ausführung der Anweisungen getestet, sodass die Wiederholung wenigstens einmal durchlaufen wird.

Beispiel

```
do {
   System.out.println("Bitte Passwort
      eingeben.");
   passwort = stringeingabe();
} while (passwort != "Sesam, öffne dich");
```

Struktogramm



Autoren: Peter Hubwieser, Patrick Löffler, Petra Schwaiger, Matthias Spohrer, Markus Steinert, Siglinde Voß, Ferdinand Winhard
Gestaltung: Andrea Eckhardt
Illustration: Andrea Eckhardt

Seite aus: Informatik 5 SB
ISBN: 978-3-12-731068-9

6 *Java*-Packages und Importe

Java verfügt über unzählige vorgefertigte Klassen und Schnittstellen. Thematisch zusammengehörende Klassen und Schnittstellen werden zu einem Paket (*package*) zusammengefasst. Die so entstehende *Java*-Bibliothek ist riesig und enthält tausende verschiedene Klassen mit unterschiedlichsten Methoden. Um sich einer dieser Klassen bedienen zu können, muss man sie in das gewünschte Projekt importieren. In *Java* funktioniert das mit dem Schlüsselwort `import`.

Syntax:

`import <paketname>.<klassenname>;`	importiert nur die gewünschte Klasse des angesprochenen Paketes
`import <paketname>.*;`	importiert sämtliche Klassen des angesprochenen Paketes

Beispiel	**Erläuterung**
`import java.util.Random;`	importiert die Klasse `Random` des Paketes `java.util`
`import java.util.*;`	importiert das vollständige Paket `java.util`

Es ist unmöglich, über die vollständige Bibliothek Bescheid zu wissen. Die *Java*-Bibliothek ist jedoch gut dokumentiert, daher gilt die *API*-Dokumentation (*Application Programming Interface*) des *Java*-Entwicklers Sun als wichtige Informationsquelle für jeden *Java*-Programmierer (http://java.sun.com/docs/).

Einige Pakete mit ausgewählten Klassen sind hier beispielhaft aufgelistet:

package	**Erläuterung**	
`java.awt`	Das Paket enthält Klassen zur Erstellung von grafischen Benutzeroberflächen und Bildern. Es stellt beispielsweise folgende Klassen zur Verfügung:	
	`Button`	erstellt einen beschrifteten Knopf
	`Canvas`	Mit der Klasse LEINWAND kann ein leerer rechteckiger Bereich auf dem Bildschirm erzeugt werden, auf dem gezeichnet werden kann.
`javax.swing`	Weiterentwicklung von `java.awt`. Beispielsweise enthält das Paket folgende Klassen:	
	`JButton`	erstellt einen beschrifteten Knopf
	`JFrame`	ein Fenster, welches weitere Komponenten aufnehmen kann
	`JLabel`	ein Bereich für einen kurzen Text, ein Bild oder beides
`java.lang`	enthält besonders wichtige Klassen, z. B.	
	`Math`	enthält Methoden für grundlegende mathematische Operationen, beispielsweise die Quadradwurzel (`sqrt`), trigonometrische Funktionen oder die Potenz (`pow`)
`java.net`	stellt Klassen für Netzwerk- und Internetapplikationen zur Verfügung	
`java.util`	stellt verschiedene nützliche Klassen zur Verfügung, beispielsweise	
	`ArrayList`	ein Feld mit variabler Länge (eine Liste)
	`Currency`	repräsentiert eine Währung
	`Random`	Mit einer Instanz dieser Klasse lassen sich Zufallszahlen erzeugen.

Autoren: Peter Hubwieser, Patrick Löffler, Petra Schwaiger, Matthias Spohrer, Markus Steinert, Siglinde Voß, Ferdinand Winhard
Gestaltung: Andrea Eckhardt
Illustration: Andrea Eckhardt

Seite aus: Informatik 5 SB
ISBN: 978-3-12-731068-9

7 Fehlerbehandlung (Exceptions)

Zur strukturierten und flexiblen Behandlung von Fehlern, die während der Programmausführung auftreten können, gibt es in *Java* eine elegante Möglichkeit: Exceptions (Ausnahmen).

Syntax:

```
try {
   <Anweisungen>
}
catch(<Ausnahmetyp> <Bezeichner>) {
   <Anweisungen>
}
```

```
try {
   <Anweisungen>
}
catch(<Ausnahmetyp> <Bezeichner>) {
   <Anweisungen>
}
finally {
   <Anweisungen>
}
```

`<Ausnahmetyp>`	Es gibt eine Vielzahl von Ausnahmen, aber alle werden von der Klasse `java.lang.Exception` abgeleitet. Es können auch eigene Exceptions programmiert werden.
`<Bezeichner>`	Der Fehlerbezeichner ist frei wählbar, oft wird einfach kurz `e` (für exception) genommen.

Im `try`-Block steht der überwachte Programmbereich, hinter `catch` folgt der Programmblock, der beim Auftreten des betreffenden Fehlers ausgeführt wird. Es können auch mehrere `catch`-Blöcke hintereinander ausgeführt werden, um unterschiedliche Fehlertypen abzufangen bzw. zu behandeln.
Da nach einer Fehlerbehandlung nicht einfach an der Stelle fortgefahren werden kann, an der der Fehler auftrat, gibt es den optionalen `finally`-Block nach einem oder mehreren `catch`. Anweisungen im `finally`-Block werden immer ausgeführt, egal ob ein Fehler von einem `catch` abgefangen wurde oder ob überhaupt ein Fehler auftrat. Sinnvoll ist das beispielsweise, wenn Dateien geschlossen oder Ressourcen freigegeben werden sollen.

Beispiele:

```
public void sicherDividieren (int z, int n) {
   int temp = -1;
      try {
      temp = z / n;
   }

   catch(ArithmeticException e) {
      System.out.println("Fehler:
           Divisor = 0.");
   }

   finally {
      System.out.println(temp);
   }
}
```

Bei einer ganzzahligen Division soll die Division durch null ausgeschlossen werden.
Im `try`-Block wird die Division versucht.

Im `catch`-Block wird untersucht, ob eine `ArithmeticException` auftritt. Dies ist genau bei einer Division durch null der Fall. Dann wird der angegebene Text auf der Konsole ausgegeben.

Zum Schluss wird auf jeden Fall noch der Wert von `temp` auf der Konsole ausgegeben.

Autoren: Peter Hubwieser, Patrick Löffler, Petra Schwaiger, Matthias Spohrer, Markus Steinert, Siglinde Voß, Ferdinand Winhard
Gestaltung: Andrea Eckhardt
Illustration: Andrea Eckhardt

Seite aus: Informatik 5 SB
ISBN: 978-3-12-731068-9

```
public void readfile() {
   RandomAccessFile f = null;
   try {
      f = new RandomAccessFile("Testfile.txt",
            "r");
      for (String line; (line = f.readLine()) !=
            null;) {
         System.out.println(line);
      }
   }
   catch(FileNotFoundException e) {
      System.out.println("Datei existiert
            nicht.");
   }
   catch(IOException e) {
      System.out.println("Fehler beim Lesen/
            Schreiben!");
   }
}
```

Dieses Beispiel benötigt das Paket `java.io.*`.
Die Klasse `RandomAccessFile` ermöglicht einen flexiblen Dateizugriff (z. B. ob nur Lesen erlaubt ist) und wird ausführlich in der *Java-API* erläutert.
Der Parameterwert `"r"` im Konstruktor erteilt nur Leserechte.
In der `for`-Wiederholung des `try`-Blocks wird die Textdatei zeilenweise ausgelesen.

Falls die angegebene Datei `"Testfile.txt"` nicht existiert, fängt der erste `catch`-Block den Fehler ab und erzeugt eine Meldung auf der Konsole.

Bei der Fehlerbehandlung gibt es in *Java* die Grundregel *catch or throw*, nach der jede Ausnahme behandelt oder weitergegeben werden muss. Jede Ausnahme, die nicht behandelt, sondern weitergegeben werden soll, muss mit dem Schlüsselwort `throws` am Ende des Methodenkopfes angegeben werden.

Syntax:

```
<Methodenrumpf>
throws <Ausnahmetyp> {
}
```

beispielsweise `public void hello(String name)` (Seite 163)
Die geschweifte Klammer öffnet sich erst nach dem `throws`.

Beispiel:

```
public boolean istPrimzahl(int n)
throws ArithmeticException {
   boolean temp = true;
   if (n <= 0) {
      throw new ArithmeticException("Fehler,
          Parameter muss > 0 sein.");
   }
   if (n == 1) {
      temp = false;
   }
   for (int i = 2; i <= n / 2; i++) {
      if (n % i == 0) {
         temp = false;
      }
   }
   return temp;
}
```

Die Methode soll untersuchen, ob eine eingegebene ganze Zahl eine Primzahl ist.

Wenn der Parameter n kleiner oder gleich null ist, wird ein neues Fehlerobjekt der Klasse `ArithmeticException` angelegt und mit der `throw`-Anweisung dazu verwendet, eine Ausnahme zu erzeugen.

Autoren: Peter Hubwieser, Patrick Löffler, Petra Schwaiger, Matthias Spohrer, Markus Steinert, Siglinde Voß, Ferdinand Winhard
Gestaltung: Andrea Eckhardt
Illustration: Andrea Eckhardt

Seite aus: Informatik 5 SB
ISBN: 978-3-12-731068-9

Einige ausgewählte Fehlertypen:

`ArithmeticException`	Beispiel: ganzzahlige Division durch null

`ArrayIndexOutOfBoundsException`	Indexgrenzen missachtet, beispielsweise bei
	`int[] a = new int[4];`
	`int b = a[7];`

`IllegalArgumentException`	tritt ein, wenn Methoden falsche Argumente melden So löst etwa `int i = Integer.parseInt("Hallo");` eine `NumberFormatException` aus.

`NullPointerException`	Dieser Fehler tritt häufig auf. Beispiel:
	`String meier = null;`
	`int i = meier.length();`

Alle möglichen Exceptions findet man in der *Java-API* (vergleichen Sie hierzu Seite 172).

8 Threads und Synchronisation

Zur Synchronisation nebenläufiger Prozesse (Threads) bedient sich *Java* des Monitorkonzepts. Dabei wird eine Menge von Attributen und Methoden zusammengefasst und gemeinsam überwacht, sodass immer nur ein Prozess auf dieser Menge arbeiten kann. In *Java* verwendet man dazu das Schlüsselwort `synchronized`. Alle derart bezeichneten Methoden einer Instanz der betreffenden Klasse werden von der Laufzeitumgebung zu einem Monitor zusammengefasst.

Beispiel:

Die Klasse `Countdown` verfügt über eine Methode zum Rückwärtszählen.

```
public class Countdown {
  int counter;

  public Countdown(int z) {
    counter = z;
  }

  public synchronized int naechsteZahl() {
    int temp = counter;
    try {
      Thread.sleep(200);
    }
    catch(Exception e){ }
    counter = counter - 1;
    return temp;
  }
}
```


Autoren: Peter Hubwieser, Patrick Löffler, Petra Schwaiger, Matthias Spohrer, Markus Steinert, Siglinde Voß, Ferdinand Winhard
Gestaltung: Andrea Eckhardt
Illustration: Andrea Eckhardt

Seite aus: Informatik 5 SB
ISBN: 978-3-12-731068-9

```
public class TZ extends Thread {

  private String name;
  private Countdown zaehler;

  public TZ(String n, Countdown z) {
    name = n;
    zaehler = z;
  }

  public static void main(String[] args) {
    Thread[] t = new Thread[4];
    Countdown z = new Countdown(50);
    for (int i = 0; i < 4; i++) {
      t[i] = new TZ("Thread-Nr." +i, z);
      t[i].start();
    }
  }

  public void run() {
    while (true) {
      System.out.println(zaehler.naechste-
          Zahl()+" - "+name);
      if(zaehler.naechsteZahl() <= -1) {
        break;
      }
    }
  }
}
```

Die Klasse `TZ` erzeugt ein Feld mit vier Threads, die alle auf denselben Countdown zugreifen.

Das Schlüsselwort `synchronized` in der Methode `naechsteZahl()` (Beispiel auf vorheriger Seite) verhindert, dass die Threads gleichzeitig die Methode verwenden können.

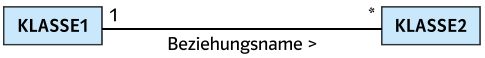
Autoren: Peter Hubwieser, Patrick Löffler, Petra Schwaiger, Matthias Spohrer, Markus Steinert, Siglinde Voß, Ferdinand Winhard
Gestaltung: Andrea Eckhardt
Illustration: Andrea Eckhardt

Seite aus: Informatik 5 SB
ISBN: 978-3-12-731068-9

UML-Überblick

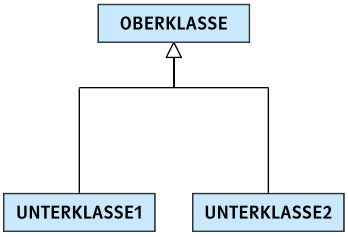
Theorie

Assoziation

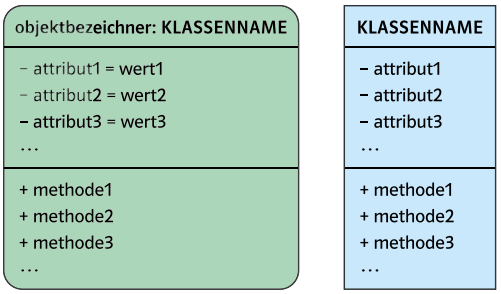


Aggregation

Vererbung

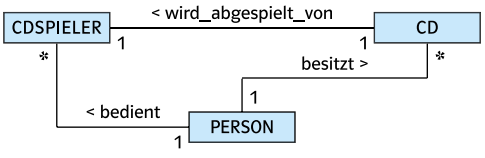


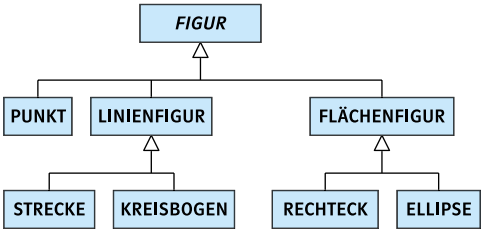
Objektkarte und Klassenkarte

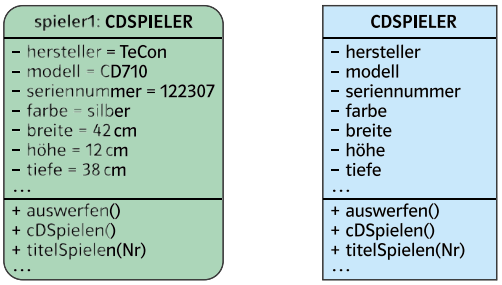


– privates Element (`private`)
+ öffentliches Element (`public`)

Beispiele





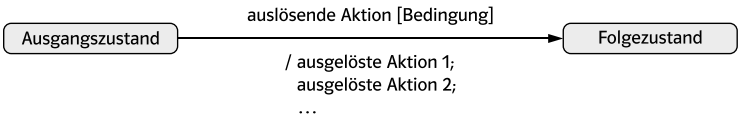


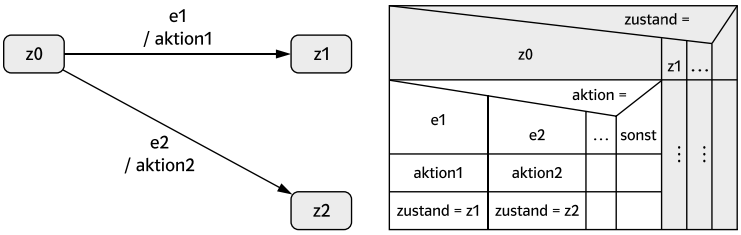
Autoren: Peter Hubwieser, Patrick Löffler, Petra Schwaiger, Matthias Spohrer, Markus Steinert, Siglinde Voß, Ferdinand Winhard
Gestaltung: Andrea Eckhardt
Illustration: Andrea Eckhardt

Seite aus: Informatik 5 SB
ISBN: 978-3-12-731068-9

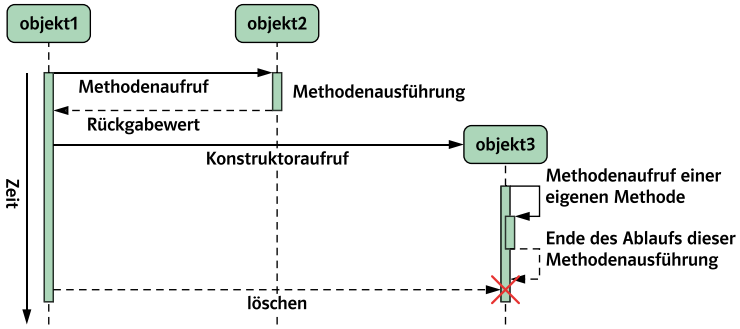
Theorie

Zustandsdiagramme

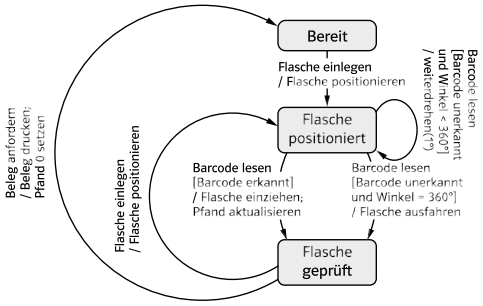


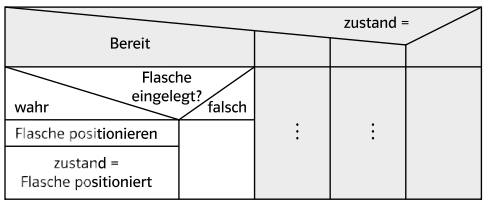


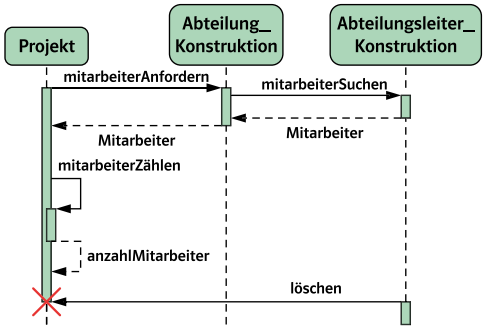
Sequenzdiagramme



Beispiele







Autoren: Peter Hubwieser, Patrick Löffler, Petra Schwaiger, Matthias Spohrer, Markus Steinert, Siglinde Voß, Ferdinand Winhard
Gestaltung: Andrea Eckhardt
Illustration: Andrea Eckhardt

Seite aus: Informatik 5 SB
ISBN: 978-3-12-731068-9