Java Grundlagen

Einführung in die Java-Programmierung

integrata
Cegos Group

inspire
qualify
change

Interaktive Dokumentation

- Sie können mit diesen Buttons ab der Seite „Inhaltsverzeichnis" (nachfolgende Seite) **navigieren**.

	Vorherige Seite, nächste Seite		*Anfang des Kapitels*
	Zurück zum Inhaltsverzeichnis		*Link zu einer anderen Seite*

Inhalts-verzeichnis

Kapitel 01: Geschichte und Charakteristik

- Historie
- Java SE Development Kit
- Java Editionen
- Charakteristika von Java
- Die Virtuelle Maschine
- Der Bytecode
- Entwicklungsumgebungen
- Komponenten der Java Standard Edition

Historie

- 1995 JDK 1.0-alpha: kostenlos über das Internet verfügbar
- 1996 JDK 1.0: in Netscape Navigator
- 1997 JDK 1.1: verbesserte APIs (JavaBeans, Events, JDBC, JNI, RMI)
- 1998 JDK 1.2: neue APIs (Collections, Swing, CORBA u. a.), Innere Klassen Performance-Verbesserungen
- 2000 JDK 1.3: Aufteilung in Standard, Enterprise und Micro Edition
- 2001 JDK 1.4: neue APIs (Logging, XML u. a.)
- 2004 JDK 5.0: neue Sprachelemente, generische Datentypen, Autoboxing, variable Parameterliste, neue APIs
- 2006 JDK 6: neue Sprachelemente und APIs, XML und Web-Services Performanceverbesserung
- 2011 JDK 7: neue APIs, Sprachverbesserungen und Spracherweiterungen
- 2014 JDK 8: Default-Implementierungen in Interfaces, Lambda Ausdrücke, neue Date&Time API …
- 2017 JDK 9: Modul-Konzept, Erweiterungen der Stream-API

Java SE Development Kit

- JRE (Java Runtime Environment) = Laufzeitumgebung
- JDK (Java Development Kit) = Software Entwicklungs-Paket
- JDK = JRE + Compiler + Tools
- Seit Version 1.2 Aufteilung in drei Plattformen
  - Java Standard-Edition (Java SE, J2SE)
  - Java Enterprise Edition (Java EE, J2EE)
    - Serverseitige Komponenten, setzt auf Java SE auf
  - Java Micro Edition (Java ME, J2ME), Wireless Toolkit
    - für Kleingeräte
- In diesem Kurs nur Java SE
  - Kostenloser Download von Oracle: http://www.oracle.com/technetwork/java/javase/downloads/index.html

Java Editionen

Micro Edition

- Java Technology Enabled Device

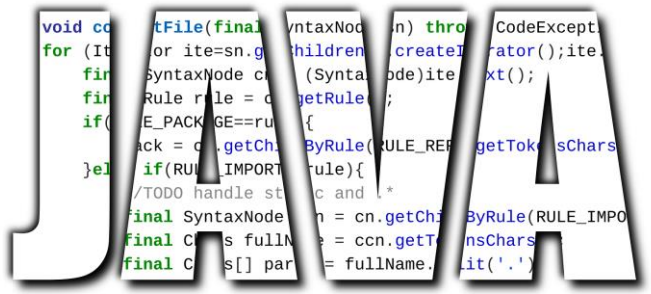
Enterprise Edition

- High-EndServer
- Baut auf Standard Edition auf

Standard Edition

- Java Technology Enabled Desktop

Charakteristika von



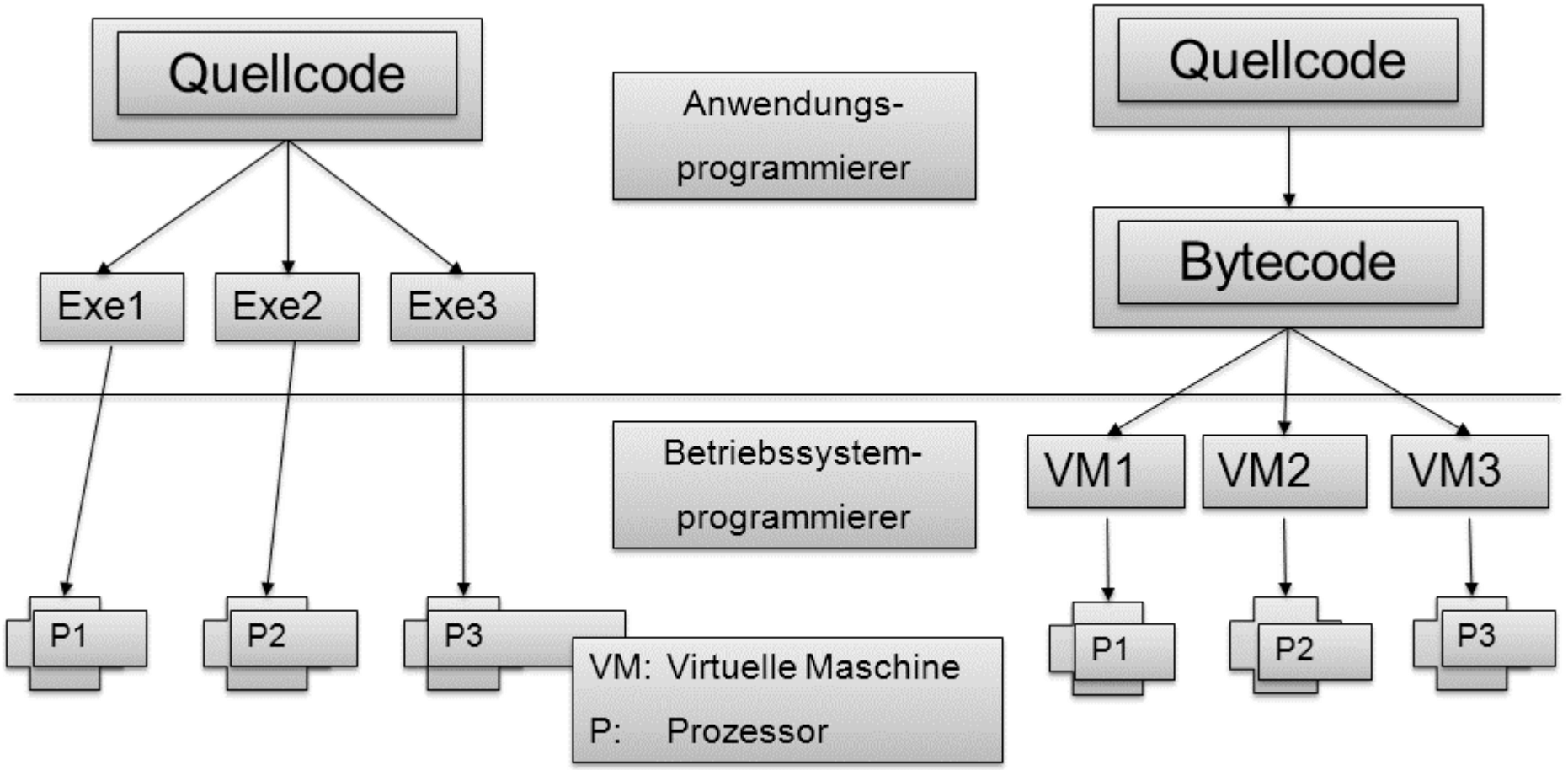
- Java ist eine einfache Programmiersprache
- Java hat einen überschaubaren Sprachumfang
- Java ist eine mit C und C++ verwandte Sprache
- Java ist eine Interpretersprache
- Java ist eine architekturneutrale Sprache
- Java ist eine rein objektorientierte Sprache
- Java erlaubt keine direkten Zugriffe auf Betriebssystemressourcen
- Java lässt robuste, stabile Programme entstehen
- Java ist eine streng typisierte Programmiersprache
- Java hat die Behandlung von Ausnahmen in Form von Exception-Klassen implementiert
- Java unterstützt nebenläufige Teilprozesse und verteilte Anwendungen

Die Virtuelle Maschine

- Funktionsweise spezifiziert ehemals von Sun
  - Implementiert von verschiedenen Herstellern
  - Virtuelle Maschinen sind für viele Plattformen bereits vorhanden

- Virtuelle Maschinen
  - Interpretieren Bytecode-Anweisungen
    - Bytecode ist ein prozessorunabhängiger Satz von maschinennahen Befehlen
  - Kontrollieren den Programmlauf auf Fehler
  - Haben eine automatische Speicherbereinigung
  - Zahlen und die Gleitkomma-Arithmetik sind plattformunabhängig gemäß dem IEEE Standard definiert
  - Einbindung externer Bibliotheken durch “Java Native Interface“ möglich
  - Integrierte Ausnahmebehandlung vorhanden
  - Das Laden benötigter Programmteile ist dynamisch zur Laufzeit möglich

Die Virtuelle Maschine



Der Bytecode

- Plattformunabhängige Interpreter-Sprache für die Virtuelle Maschine
  - Ähnlichkeiten zu Assembler
- Spezifikation enthält Sicherheits-Mechanismen
  - Korrektheit des Bytecode-Formats
  - Übereinstimmung der Typen bei Zuweisungsoperationen und Aufrufen von Routinen
  - Kein Zugriff auf nicht-initialisierte Speicherbereiche oder Variablen
  - Keinerlei direkter Speicherzugriff

Classloader:

- Unterscheidung Systemklassen und Anwenderklassen
- Lokation der Bytecode-Dateien

SecurityManager:

- Aufsetzen einer Sicherheitsumgebung für den Anwender
- Dateibasierte Konfiguration

Der Klassenpfad

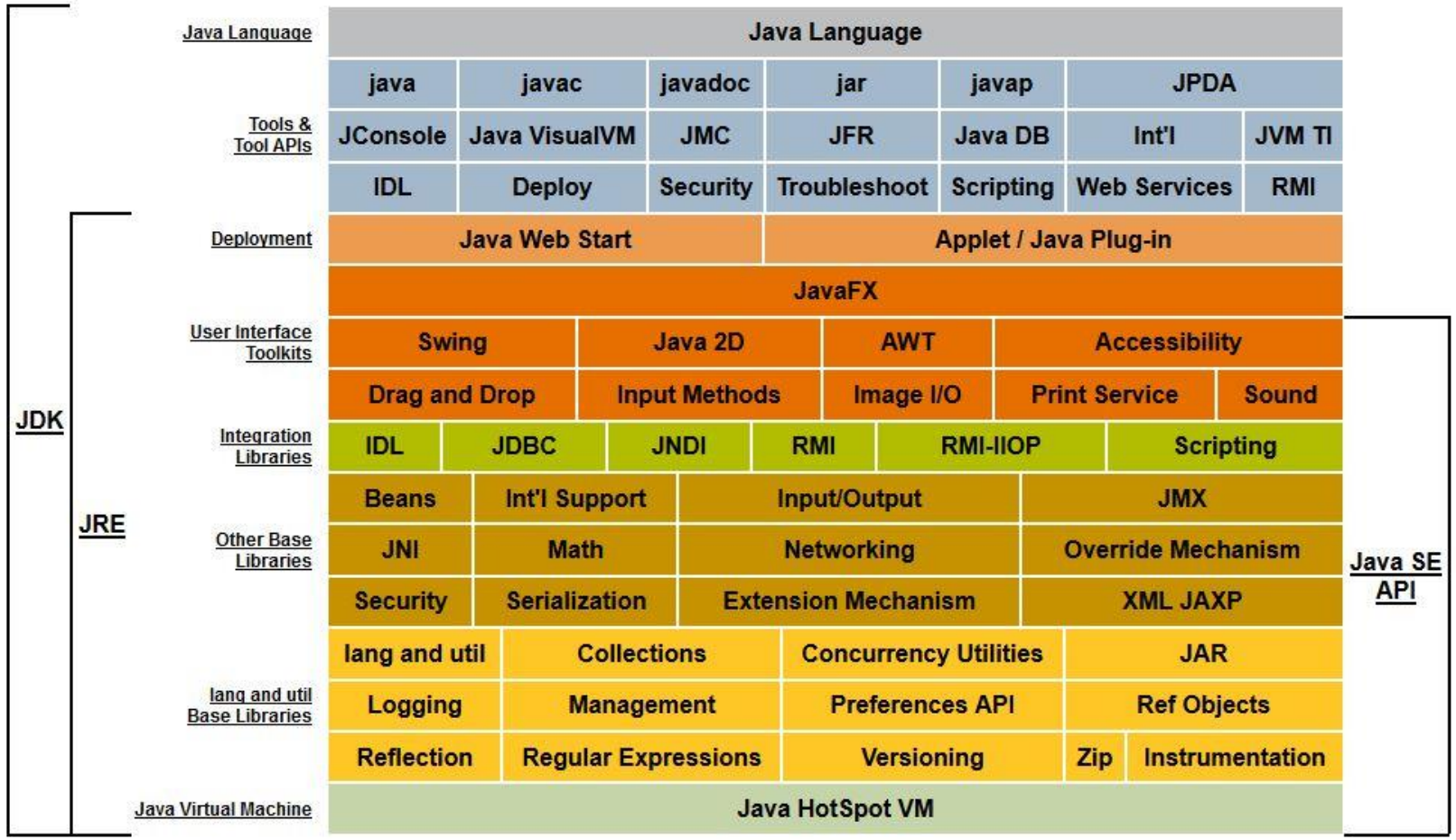
- Die Lokation der Java-Archive und der Klassen wird von drei Quellen bestimmt
  - Systemklassen: Datei **rt.jar** in **%JAVA_HOME%\jre\lib**
  - Erweiterungsklassen: Java-Archive in **%JAVA_HOME%\jre\lib\ext**
  - Anwendungsklassen
    - Pfadangaben in der Umgebungsvariablen **CLASSPATH**
    - Übergabeparameter **-cp** beim Start der Virtuellen Maschine
- Standard (wenn CLASSPATH nicht gesetzt):
  - `.` (Punkt) = aktuelles Working-Directory, Pakete in Unterdirectories
  - Datei `jre/lib/rt.jar` = Klassenbibliothek der verwendeten Java-Software, liegt relativ zum `bin`-Directory (`../jre/lib`)
  - alle jar-Dateien in `jre/lib/ext`
- CLASSPATH setzen:
  - `CLASSPATH` enthält: Directories (deren Unterdirectories die Pakete sind), jar-Dateien (die eine Hierarchie von Paketen enthalten), durch Semikolon (Windows) oder Doppelpunkt (Unix) getrennt

Entwicklungs-umgebungen

- Eclipse von der Eclipse Foundation
- NetBeans von Oracle Corporation
- BlueJvon der BlueJ Group (University of Kent)
- IntelliJ IDEA von JetBrains

u. v. a. m.

Komponenten der Java Standard Edition



© https://docs.oracle.com/javase/8/docs/

Kapitel 02: Grundlagen der Java- Programmierung

- Java Quellcode
- Einfaches Beispiel
- Ausgaben
- Einfache Datentypen
- Deklaration von Variablen
- Finale Variablen
- Konstanten

Java Quellcode

- Namenssyntax - Bezeichner werden gebildet aus:
  - Buchstaben
  - Ziffern
  - Unterstrich
- Kommentare
  - Zeilenkommentar: `//`
  - Blockkommentar:

```
/*
   Block-Kommentar
*/
```

  - Dokumentationskommentare:

```
/**
   Javadoc-Kommentar
*/
```

Einfaches Beispiel

- Java-Code in der Datei HelloWorld.java

```
/* Mehrzeiliger Kommentar
 * Die Klasse HelloWorld soll den Text "Hello World"
 * auf den Monitor ausgeben.
 */
class HelloWorld
{ // Beginn der Klasse
  public static void main( String[] args )
  { //Beginn main

    System.out.println( "Hello World" );

  } //Ende main
} // Ende Klasse HelloWorld
```

Ausgaben

- ohne Zeilenumbruch
  - `System.out.print( Ausdruck )`
- mit Zeilenumbruch
  - `System.out.println( Ausdruck )`
- ohne Zeilenumbruch mit Formatierung
  - `System.out.printf( Format, Ausdruck )`
  - **Formatangabe:**

%d	ganze Dezimalzahl
%e	Gleitkommazahl im Gleitkommaformat (mit Exponent)
%f	Gleitkommazahl im Festkommaformat (ohne Exponent)
%s	String

Flags:

0: Führende 0
–: linksbündig
+: immer mit Vorzeichen
blank: Führende Leerzeichen

Ausgabebeispiele

```
System.out.print( "München" );
System.out.print( "Berlin" + " " + "ist groß");
```

MünchenBerlin ist groß

```
System.out.println( "München" );
System.out.println( "Berlin" + " " +
                                "ist groß" );
```

München
Berlin ist groß

```
System.out.printf( "%5.2F", 12.345 );
```

12,35

Einfache Datentypen

- Ganzzahltypen

Typ	Größe	von	bis
byte	1 Byte	-128	127
short	2 Byte	-32768	32767
int	4 Byte	-2.147.483.648	2.147.483.647
long	8 Byte	-9.223.372.036.854.775.808	9.223.372.036.854.775.807

- Gleitpunkttypen

float	4 Byte	ungef. -3.4E+38	3.4E+38
double	8 Byte	ungef. -1.7E+308	+1.7E+308

- Nichtnumerische Typen

char	2 Byte		
boolean	1 Byte	**true**	**false**

Deklaration von Variablen

```
typ bezeichner;

typ bezeichner  =  Initialwert;
```

Beispiel:

```
boolean test;
int zahl, i = 0, j = 1;
double d = 0.0;
char zeichen = 'R';
String name;
```

Finale Variablen

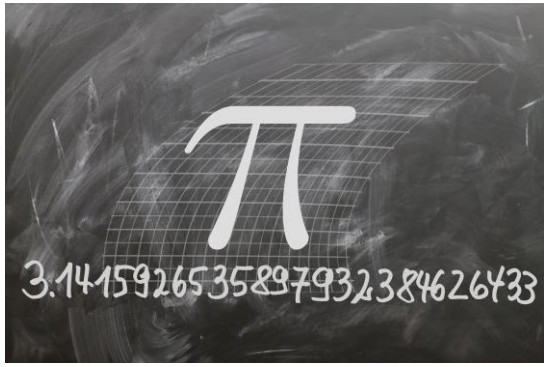
```
final typ bezeichner = Wert;
```

Beispiel:

```
final double PI = 3.14159;
```

oder

```
final double PI;
//...
PI = 3.14159;
```



Finale Variablen können nach ihrer Initialisierung nicht mehr geändert werden:

```
final double PI = 3.14159;
PI = 2.176;      // falsch: Neuzuweisung eines Wertes
```

Konstanten

- Boolesche Konstanten
  true false
- Numerische Konstanten
  Ganzzahlkonstanten
  83 033 0XAF8B 0B101
  Gleitpunktkonstanten
  23.779 5.1E6
- Zeichenkonstanten
  ‘B‘ ‘\0‘ ‘\n‘
- Stringkonstanten
  “Huber“

Kapitel 03: Operatoren und Anweisungen

- Operatoren
- Cast-Operator und ternärer Operator
- Fallunterscheidung mit if
- Fallunterscheidung mit switch
- Die while-Schleife und do-while-Schleife
- Die for-Scheife
- Die forEach-Schleife
- Sprunganweisungen und instanceof
- Array

Operatoren

- Zuweisungsoperator
  =
- Arithmetische Operatoren

a + b	Addition
a – b	Subtraktion
a * b	Multiplikation
a / b	Division
a % b	Modulo-Division (nur für Ganzzahlobjekte erlaubt)

- Kurzschreibweise
  a += b
  a -= b
  a *= b
  a /= b
  a %= b
- Inkrement und Dekrement
  a++ oder ++a
  a-- oder --a

Operatoren

- Vergleichsoperatoren

  `>` `<` `>=` `<=` `!=` `==`

- Logische Operatoren

  `&&` Und (binär)

  `||` Oder (binär)

  `!` Nicht, logische Negation (unär)

Operatoren: Cast-Operator und ternärer Operator

- Explizite Typumwandlung (Cast-Operator)

  Syntax:

  (typ) Ausdruck

  Beispiel:

```
char a = (char) 65;
```

- Bedingte Bewertung (ternär)

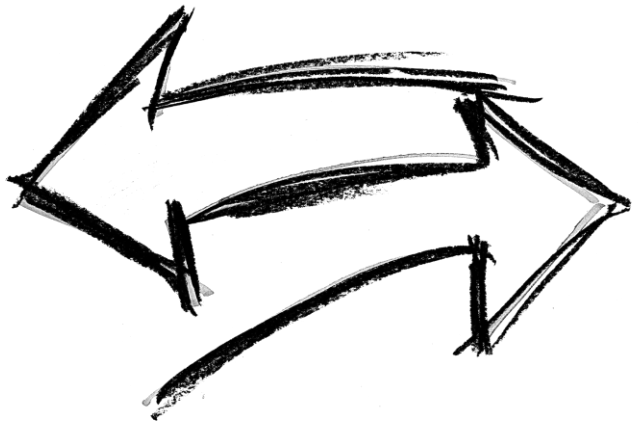
  Syntax:

```
boolean ? Wert1 : Wert2
```

  Beispiel:

```
min = (a < b) ? a : b;
```

Anweisungen: Fallunterscheidung mit if



- Syntax:

```
if (boolescher Ausdruck)
     Anweisung
```

- Syntax mit else:

```
if (boolescher Ausdruck)
     Anweisung
else
     Anweisung
```

- Syntax alternativ mit Block:

```
if (boolescher Ausdruck){
     Anweisungen
     Anweisungen
}
else {
     Anweisungen
     Anweisungen
}
```

Anweisungen: Fallunterscheidung mit switch

- Syntax:

```
switch (ganzzahliger, char-Ausdruck oder String ) {
      case Konstante1:
            Anweisung1.1
            Anweisung1.n
      case Konstante2:
            Anweisung2.1
            Anweisung2.n
      default:
            Anweisungd.1
            Anweisungd.n
}
```

Anweisungen: while-Schleife und do-while-Schleife

- Syntax while-Schleife: (kopfgesteuert)

```
while (boolescher Ausdruck)
      Anweisung
```

- Syntax do-while-Schleife: (fußgesteuert)

```
do
      Anweisung
while (boolescher Ausdruck);
```

- Syntax alternativ mit Block:

```
while (boolescher Ausdruck){
      Anweisungen
      Anweisungen
}
```

- Syntax alternativ mit Block:

```
do {
      Anweisungen
      Anweisungen
}while (boolescher Ausdruck);
```

Anweisungen: for-Schleife

- Syntax:
(kopfgesteuert)

```
for ( Ausdruck1, Ausdruck2, Ausdruck3 )
            Anweisung
```

- Syntax mit Block:

```
for ( Ausdruck1, Ausdruck2, Ausdruck3 ) {
            Anweisungen
            Anweisungen
}
```

- Ausdruck1: Wird nur einmal zu Beginn der Schleife bewertet (Initialisierung).
- Ausdruck2: Muss ein boolescher Ausdruck sein und stellt die Bedingung dar.
- Ausdruck3: Wird nach jedem Schleifendurchlauf am Ende bewertet (Re-Initialisierung).

Anweisungen: forEach-Schleife

- Syntax:
  (kopfgesteuert)

```
for ( typ Element : Sammlung )
            Anweisung
```

- Syntax mit Block:

```
for ( typ Element : Sammlung ) {
            Anweisungen
            Anweisungen
}
```

- typ: Kann ein einfacher oder komplexer Datentyp sein.
- Element: Variablenname für das einzelne Element der Sammlung.
- Sammlung: Kann ein Array oder eine andere Collection sein.

Anweisungen: return und instanceof

- Sprunganweisung

  Syntax:

```
break;
continue;
```

- return

  Syntax:

```
return Ausdruck;
return;
```

- instanceof Operator

  Syntax:

```
referenz instanceof  Klasse
```

Komplexer Datentyp Array

- Syntax:

```
typ [ ] arrayName;
```

Beispiel:

```
int [ ] array1;
int [ ] array2 = new int [ 5 ];
```

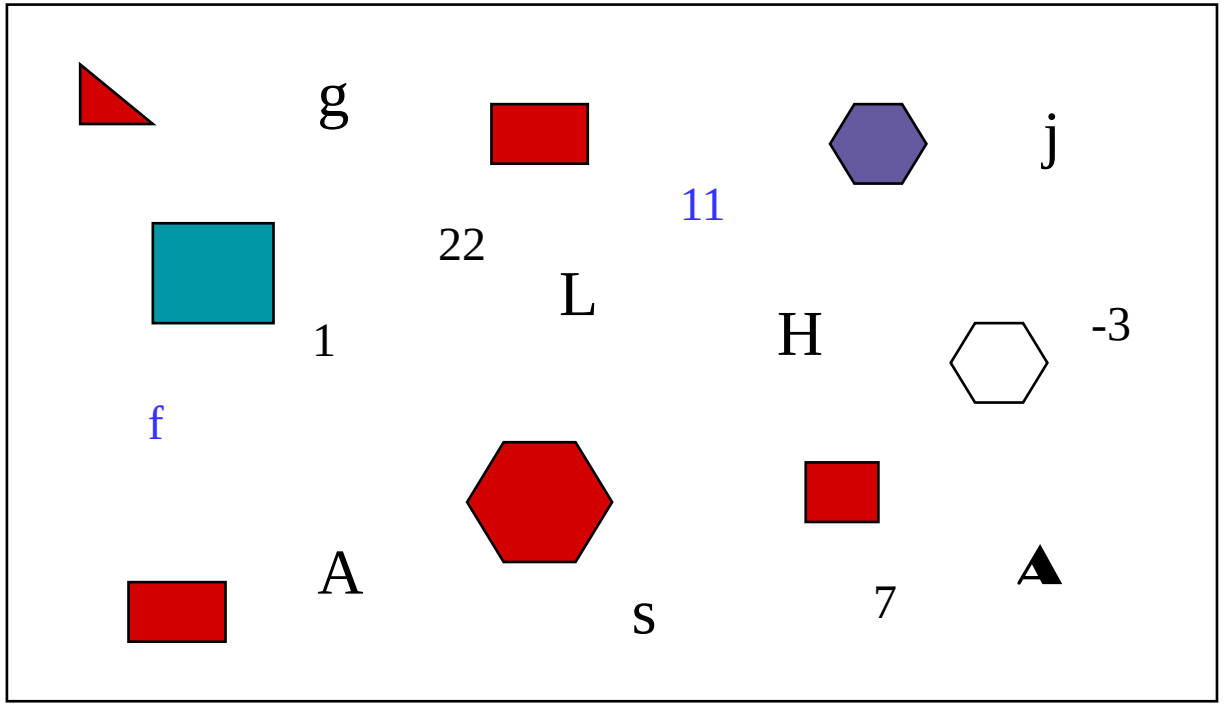
Mit der Definition eines Arrays wird ein Feld namens length erzeugt, welche als Wert die Anzahl der Elemente enthält. Das Längenfeld kann über arrayname.length angesprochen werden.

```
array2.length
```

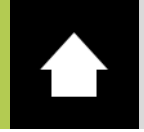
Kapitel 04: Objektorientierte Programmentwicklung

- Das Problem
- Der objektorientierte Ansatz
- Das Objekt
- Die Klasse
- UML-Klassendiagramm
- Klassendefinition
- Überladen von Methoden
- Konstruktoren
- Klassenattribute und Klassenmethoden
- Ausnahmen

Das Problem



Komplexes, ungeordnetes System, Zusammenhänge?
Was ist wesentlich, was unwesentlich?
Existieren Abhängigkeiten?

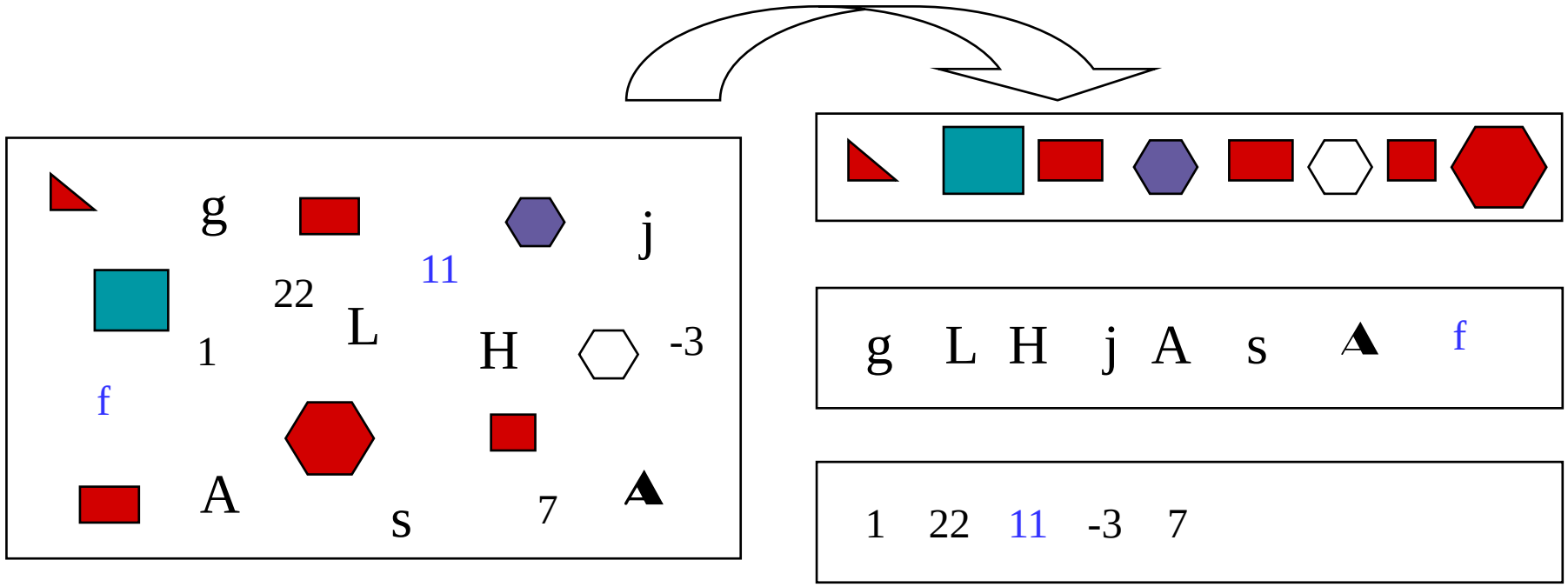


Der objektorientierte Ansatz

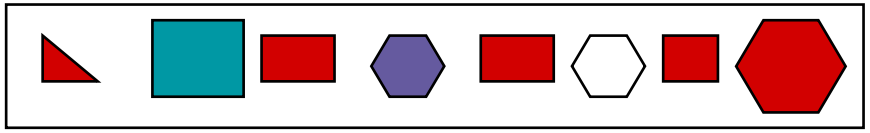
- Klassifizieren
- Abstrahieren
- Ordnen, Bilden von Hierarchien

- Ein „menschlicher“ Lösungsansatz!

Klassifizieren



Abstrahieren



g L H j A s f

1 22 11 -3 7

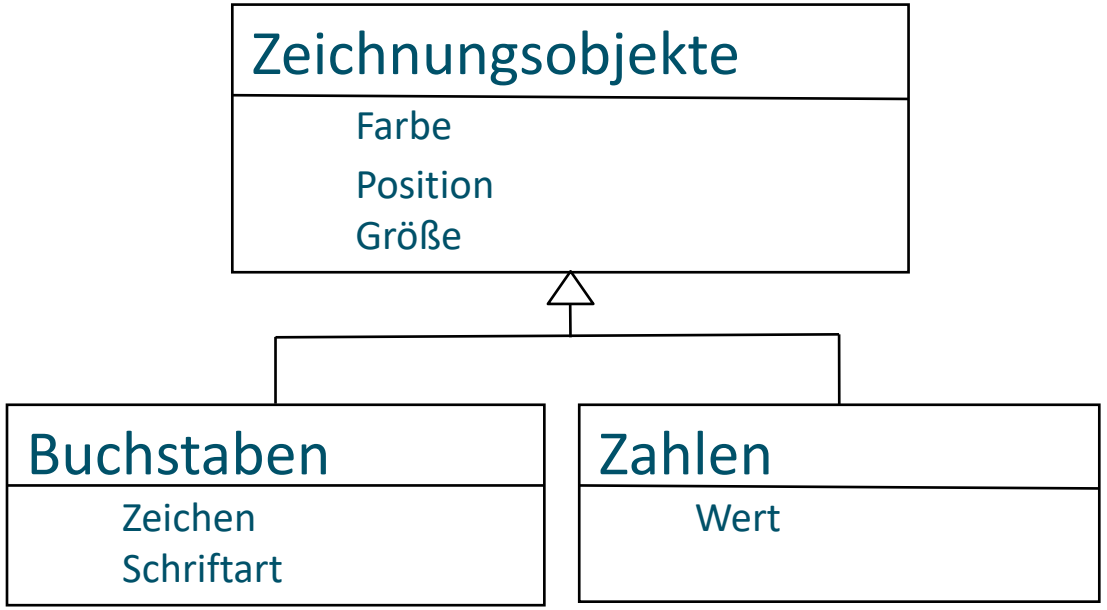
Zeichnungsobjekte
- Farbe
- Position
- Größe

Buchstaben
- Zeichen
- Schriftart
- Farbe
- Position
- Größe

Zahlen
- Wert
- Farbe
- Position
- Größe

Ein Zeichnungsobjekt **hat eine** Farbe, eine Position und eine Größe als Eigenschaften.
Das komplexe Zeichnungsobjekt ist eine **Komposition** einfacherer Elemente.

Ordnen, Bilden von Hierarchien



Ein Buchstabe **ist ein** Zeichnungsobjekt, das ein Zeichen in einer Schriftart darstellt.
Eine Zahl **ist ein** Zeichnungsobjekt, das einen Zahlenwert darstellt.

Buchstaben und Zahlen sind Zeichnungsobjekte und erben automatisch auch alle Eigenschaften eines Zeichnungsobjekts.

Was ist ein Objekt?

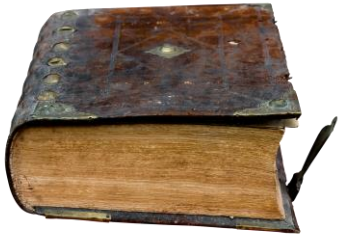
ganz konkrete

- **Gegenstände,**
- **Geräte,**
- **Ereignisse,**
- **Strukturen,**
- **Rollen,**
- **Örtlichkeiten,**

eben alles,

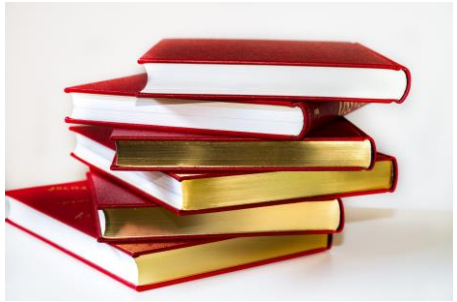
wovon man sich einen

Begriff machen kann



Ein Objekt

- **ist die Abstraktion eines “Begriffs”,**
- ***hat* eine eigene Identität,**
- **zeigt ein für seine Art typisches Verhalten,**
- **hat zu jedem Zeitpunkt einen bestimmten Zustand, der für das Verhalten in bestimmten Situationen ausschlaggebend sein kann.**



Objekt

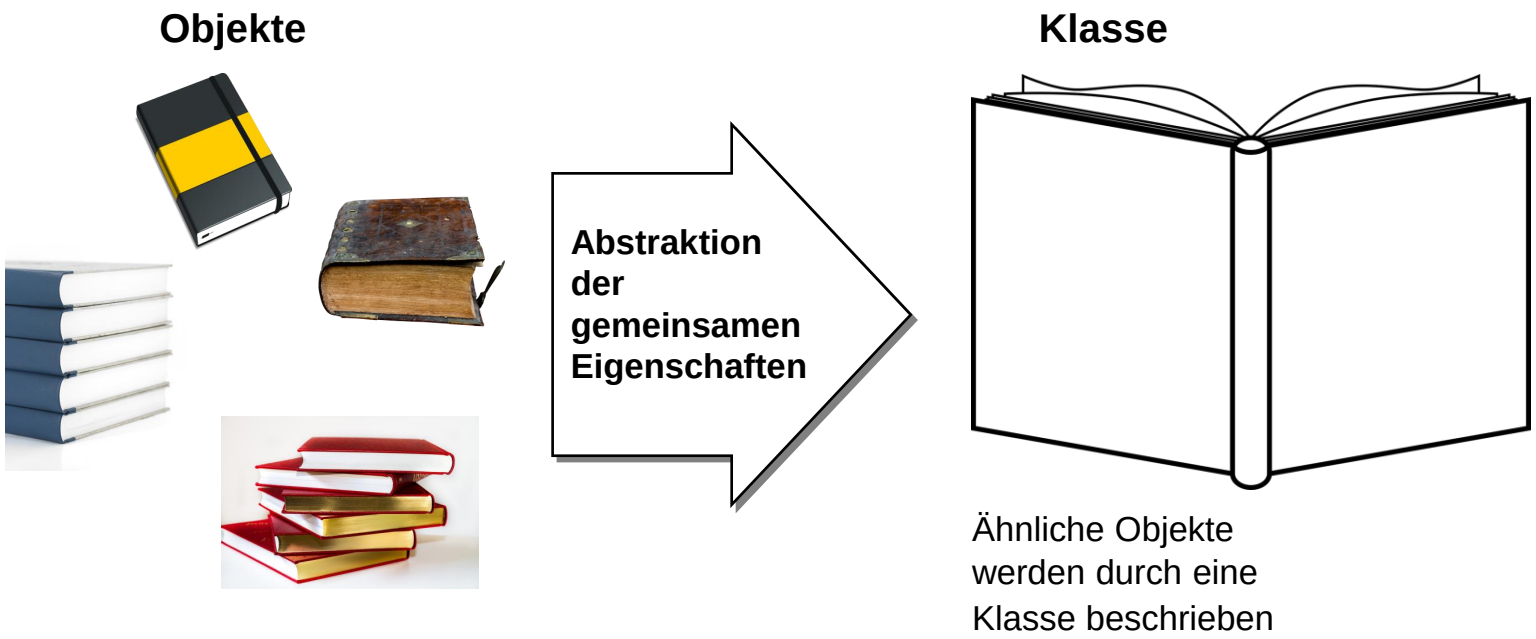
- Ein Objekt besitzt
  - Eigenschaften ⇒ Attribute
  - Fähigkeiten ⇒ Methoden
  - Interaktivität ⇒ Botschaften
- Analogie zu traditionellen Programmen
  - Attribute ⇔ Variable
  - Methoden ⇔ Funktionen, Prozeduren
  - Botschaften ⇔ Ablaufsteuerung, Parameter
- Anderer Denkansatz:
  - Ein Objekt ist stets das Ergebnis einer Produktion.
  - **Objekte werden aus der Klasse erzeugt = instanziiert**

Ein Objekt ist das Ergebnis eines reproduzierbaren Produktionsprozesses.

Ein Objekt ist eine Instanz (ein Exemplar) einer Klasse.

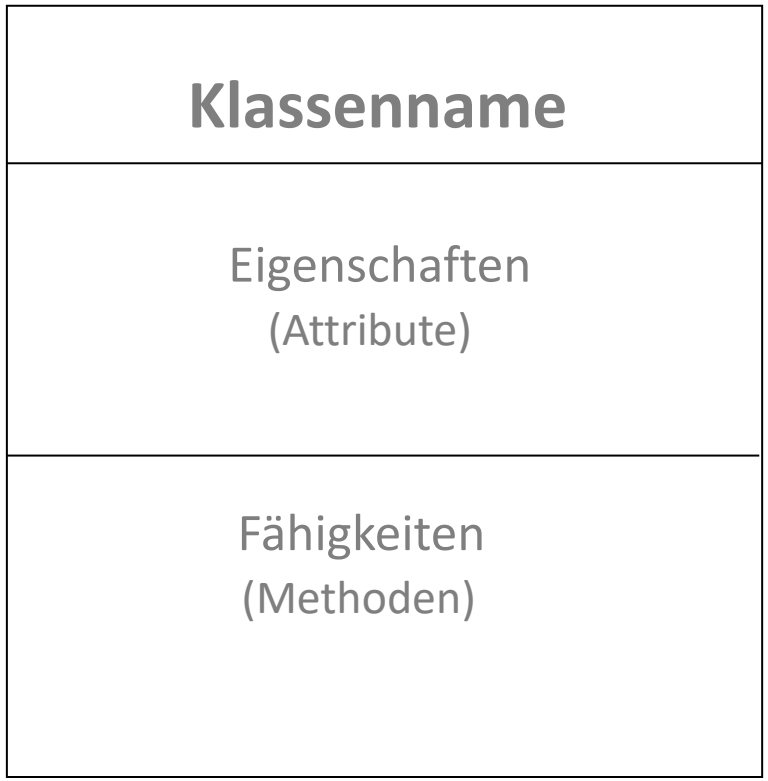
Klasse

- Gleichartige Objekte werden zu Klassen abstrahiert
  - Eine Klasse dient als Vorlage, Bauanleitung für (mehrere) Objekte

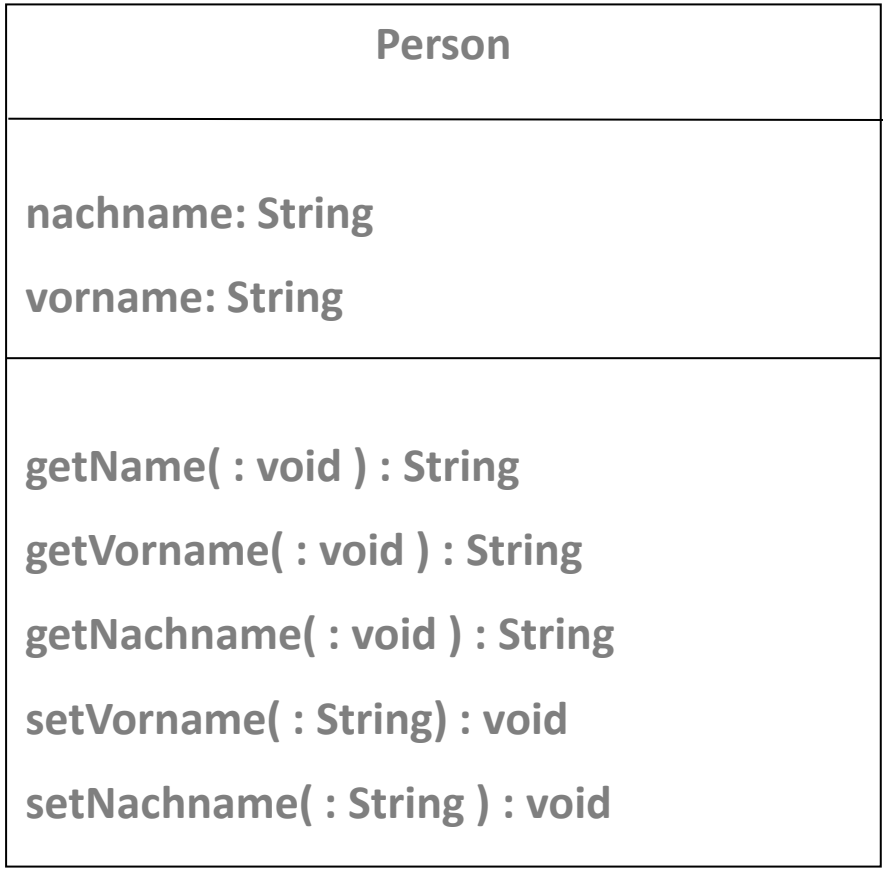


UML-Klassendiagramm

- Bildhaft lässt sich eine Klasse in Form eines Klassendiagramms, ein Bestandteil der sogenannten UML-Notation (Unified Modelling Language) darstellen:



Die Klasse Person



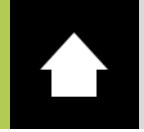
Klassendefinition

- Syntax der Klassendefinition:

```
sichtbarkeit class Klassenname {
      // Klassenkörper mit
      // Methodendefinitionen und Attributen
}
```

- Sichtbarkeit:

`public`	Sind auch außerhalb des Paketes sichtbar
`<default>`	Sind nur innerhalb des Paketes sichtbar



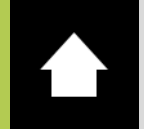
Attributdeklaration

- Syntax der Attributdefinition:

```
sichtbarkeit typ attributname;
```

- Sichtbarkeit:

`private`	Sind nur innerhalb der Klasse sichtbar
`public`	Sind auch außerhalb des Paketes sichtbar
<default>	Sind nur innerhalb des Paketes sichtbar



Methoden (Funktionalitäten)

- Syntax der Methodendefinition:

```
sichtbarkeit returntyp methodenname (typ1 par1, typ2 par2, … )
{ // methodenblock (-körper)
      mit lokalen Variablen
      und Anweisungen
}
```

- Sichtbarkeit:

private	Sind nur innerhalb der Klasse sichtbar
public	Sind auch außerhalb des Paketes sichtbar
<default>	Sind nur innerhalb des Paketes sichtbar

Überladen von Methoden

- Eine Form der Polymorphie in der Objektorientierten Programmierung wird als Überladen von Funktionen bzw. Methoden bezeichnet.

Beispiel:

```
int summe(int zahl1, int zahl2){
        return zahl1 + zahl2;
}

double summe(double zahl1, double zahl2){
        return zahl1 + zahl2;
}

int summe(int zahl1, int zahl2, int zahl3){
        return zahl1 + zahl2 + zahl3;
}
```

Die main-Methode

- Der Start eines jeden Java-Programmes beginnt mit dem Aufruf der Methode main( ).

- Beispiel: Java-Code in der Datei HelloWorld.java

```
public class HelloWorld
{
    public static void main( String[ ] args )
    { //Beginn main
        System.out.println( "Hello World" );
    } //Ende main

} //Ende Klasse HelloWorld
```

Die Methode println( )

- Monitorausgaben werden mit der Methode println( ) realisiert.

Beispiel:

```
System.out.println(1 + 2);      // Ausgabe: 3

System.out.println( "Erg=" + 1 + '2' );
                                // Ausgabe: Erg=12

System.out.println( "1" + 2 ); // Ausgabe: 12

System.out.println( 1 + '2' ); // Ausgabe: 51
```

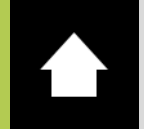
Referenzen und Instanzerzeugung

- Beispiel:

```
Person p1;          // Deklaration: p1 ist eine
                    // Person-Referenz

p1 = new Person();// Instanzerzeugung eines
                  // Objektes vom Typ Person

Person p2 = new Person();
```

- Nicht initialisierten Objektreferenzen sollte die Null-Referenz mit dem Schlüsselwort null zugewiesen werden.

```
p1 = null;
```



Zugriff auf Attribute und Methoden

- Punktoperator

  Syntax:

  ```
  referenzname.attributname;
  referenzname.methodenname( );
  ```

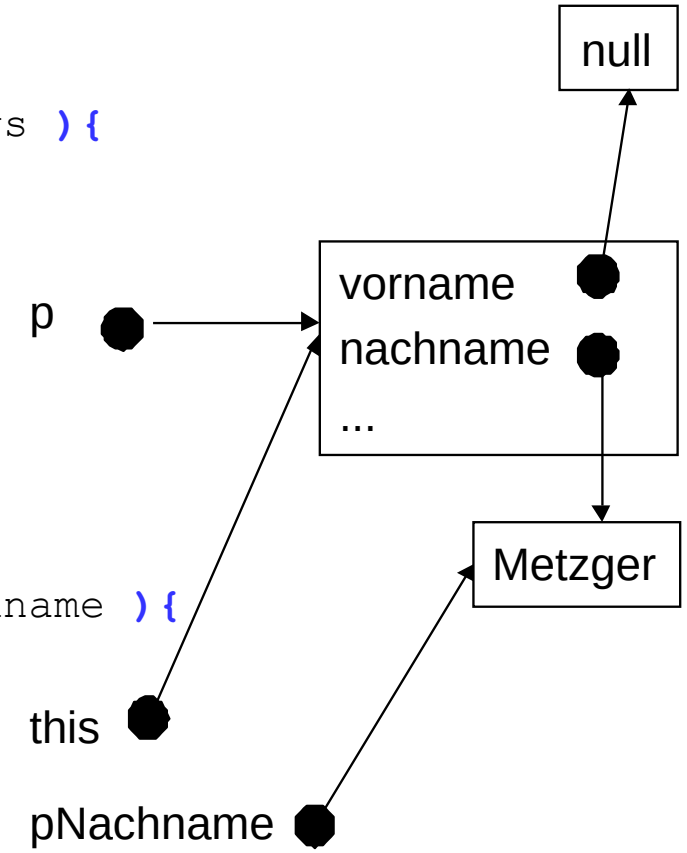
- Die this-Referenz

  Die Referenz this repräsentiert innerhalb einer Methode das aktuell gültige Objekt und kann für den Zugriff auf die Attribute und Methoden der Klasse verwendet werden.

Referenzen

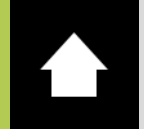
```
public class Application {
  public static void main( String[] args ){
    Person p = new Person( );
    p.setNachname( "Metzger" );
  }
}
```

```
public class Person {
  public void setNachname( String pNachname ){
    this.nachname = pNachname;
  }
}
```



Konstruktoren

- Ein Konstruktor ist eine spezielle Methode zur Erzeugung von Instanzen. Der Name eines Konstruktors ist mit dem Namen der Klasse identisch. Konstruktoren haben keinen return-Wert, sie sind typlos.
- Ein Konstruktor wird automatisch bei der Instanzerzeugung eines Objekts aufgerufen, also bei der Verwendung von new. Konstruktoren verhalten sich ansonsten wie andere Methoden.
- Der Konstruktor ohne Parameter wird default- bzw. Standard-Konstruktor genannt.
- Da Methoden überladen werden können, gilt dies natürlich auch für Konstruktoren. Eine Klasse kann also beliebig viele Konstruktoren besitzen.
- Konstruktoren können sich auch gegenseitig mit Hilfe der this-Referenz aufrufen, jedoch muss der Aufruf als erste Anweisung innerhalb des Konstruktors stehen.



Klassenattribute und Klassenmethoden

- Klassenattribute sind nicht instanzbezogen und existieren nur ein Mal pro Klasse.

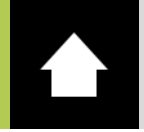
  Syntax:

```
sichtbarkeit static typ name = Wert;
```

- Klassenmethoden werden oft als Zugriffsmethoden für Klassenattribute verwendet.
  Statische Methoden können mit dem Namen der Klasse vor dem Punkoperator aufgerufen werden.

  Syntax:

```
sichtbarkeit static returntyp methodenname( param )
{
  // Zugriff nur auf statische Attribute der Klasse
}
```



Initialisierungen von Klassenattributen

- Klassenvariablen können bei ihrer Definition initialisiert werden.
- Die Initialisierung der statischen Attribute erfolgt beim Laden der Klasse.
- Zur Initialisierung von statischen Größen kann ein statischer Initialisierungsblock verwendet werden:

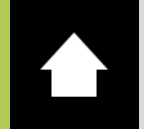
  Beispiel:

```
static private boolean a;

static private double b;


static
{
        a = true;

        b = 1.0;

}
```

- Ein `static`-Block darf keine `Exception` werfen.

Ausnahmen

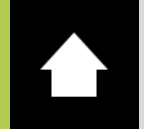
- Ein Ereignis, das während des Ablaufs eines Programms den normalen Fluss der Instruktionen unterbricht.
  - Hardware-Fehler
  - Programmierfehler
- Wenn eine Ausnahme eintritt, wird ein Objekt der Klasse **Exception** erzeugt, das zusätzliche Information enthält, die an die aufrufende Methode übergeben wird.
  - Trennung der Fehlerbehandlung vom Rest des Programms
  - Ausnahmen können dem Aufrufer standardisiert zurückgemeldet werden
    - Informationen werden in einem **Exception**-Objekt übermittelt
  - Ausnahmen können sauber durch Typen gruppiert werden



Ausnahme-behandlung: Beispiel

- Pseudocode zum Lesen einer Datei ohne Fehlerbehandlung

```
readFile {
  Öffne die Datei;
  Bestimme die Größe;
  Belege Speicher;
  Lies die Datei in den Speicher;
  Schließe die Datei;
}
```



Ausnahmen: Traditionelle Fehlerbehandlung

```
FehlerTyp readFile {
  //initialisiere Fehlerwert =0;
 //Öffne die Datei...
  if (Datei ist offen) {
     //Bestimme die Größe;
     if (Größe bestimmt) {
         //Belege Speicher;
       if(Speicher belegt) {
         Lies die Datei in den Speicher;
         if(Fehler aufgetreten)
            Fehlerwert = -1;
         } else
            Fehlerwert = -2;
       } else
         Fehlerwert = -3;
         Schließe die Datei;
     Fehlerwert = -5;
  return Fehlerwert;
}
```

Exception Schlüsselwörter

- `try`
  - Definiert einen Block, innerhalb dessen Ausnahmen (Exceptions) auftreten können (geworfen werden können).
- `catch`
  - Definiert einen Block, der die Fehlerbehandlung für die durch den im catch-Befehl angegebenen Ausnahmetyp durchführt.
- `finally`
  - Definiert einen Block, der stets ausgeführt wird, egal ob ein Fehler auftrat oder nicht. Auch wenn
  - innerhalb des catch-Zweiges eine weitere Exception geworfen wird.
  - im catch-Zweig ein return steht.
  - kein catch-Zweig durchlaufen wird.
- `throw`
  - Erzeugt (wirft) eine Ausnahme. Hierfür muss dem Befehl throw ein Objekt übergeben werden, das eine Unterklasse von Throwable ist (dies sind die Klassen Exception und Error).
- `throws Exception`
  - Die Methode muss mit throws eine Liste aller Ausnahmen definieren, die geworfen werden können.

Exception Beispiel: Kehrwert

- Folgendes Beispiel (Berechnung des Kehrwerts) soll die Verwendung der Klasse Exception verdeutlichen:

```
public class Rechnen {
  public static void main( String [ ] args) {
    try {       …
      MathLib ml = new MathLib();
      double ergebnis = ml.kehrwert( a );
      System.out.println(ergebnis);
    }
    catch (Exception e) {
      System.out.println("Kehrwert nicht berechnet");
    }
  }
}
```

```
public class MathLib {
  public double kehrwert(double d)
                    throws Exception {
     if(d == 0.0)
       throw new Exception();
     else   // else kann auch fehlen
       return 1.0/d;
  }
}
```

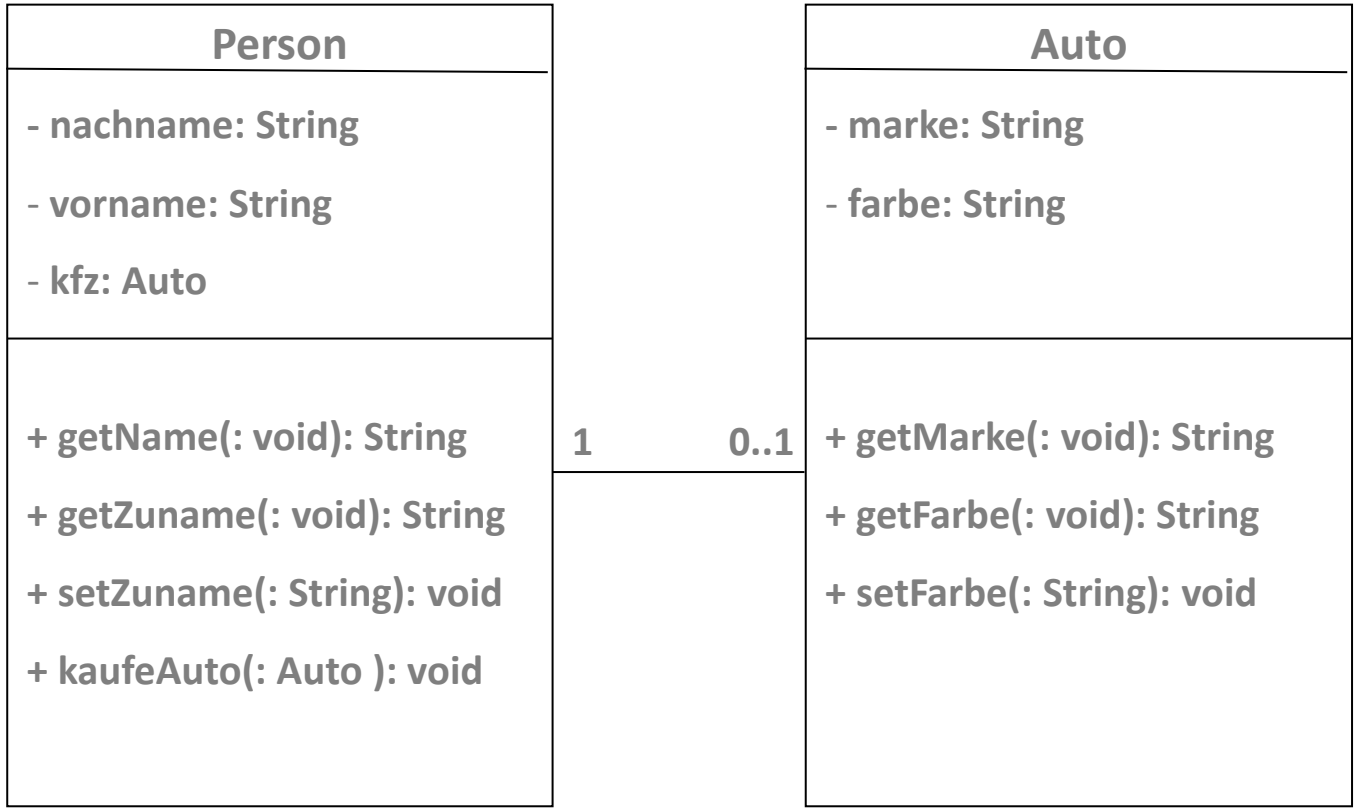
1. Aufruf: Kehrwert wird ohne Probleme berechnet (double a = 5.0;)
2. Aufruf: Kehrwert kann nicht berechnet werden (double a = 0.0;)

Kapitel 05: Beziehungen

- Assoziation
- Aggregation/Komposition
- Vererbung
- Sichtbarkeit in der Vererbungshierarchie
- Vererbung und Konstruktoren
- Vererbung und Polymorphie
- Vererbung und der Cast-Operator
- Vererbungshierarchie der Exceptionklassen
- Finale Elemente

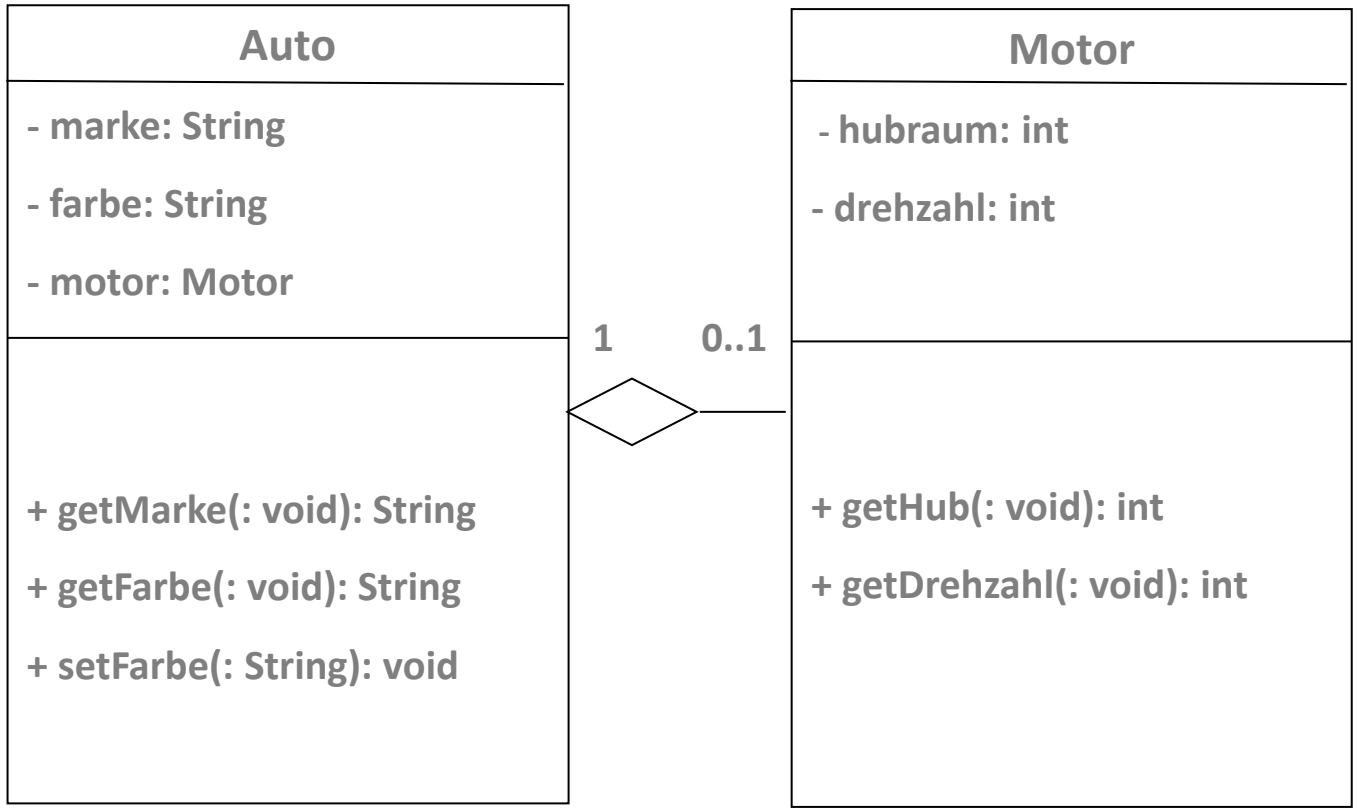
Assoziation

- Eine Formulierung wie „verwendet ein“ oder „nutzt ein“ deutet auf eine Assoziation hin.



Aggregation/ Komposition

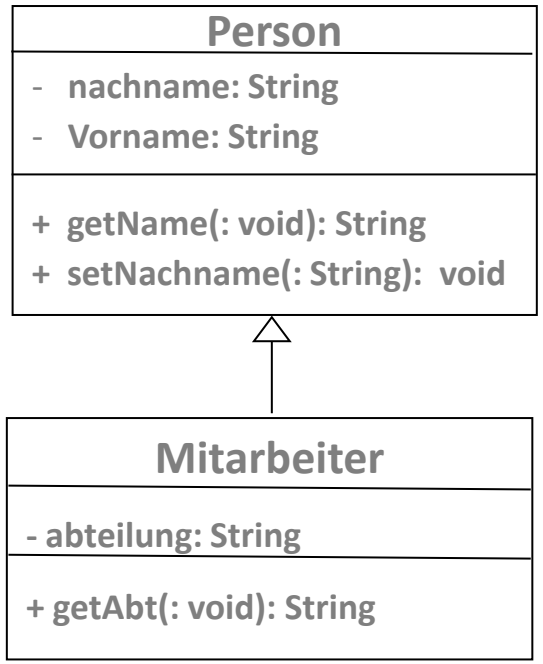
- Eine Aggregationsbeziehung liegt meist dann vor, wenn von „hat ein“ die Rede ist. Das Auto hat einen Motor.



Vererbung

- Bei der Vererbung handelt es sich um eine sogenannte „ist ein“ Beziehung.

  Ein Mitarbeiter „ist eine“ Person.



Vererbung

Syntax:

```
sichtbarkeit class Subklasse extends Superklasse{

   // Klassenkörper mit Attributen und Methodendefinitionen

}
```

Beispiel:

```
public class Mitarbeiter extends Person {

    // Attribute:
   private String abteilung;

    // Methoden:
   public String getAbteilung( ) {
     // Funktionalität:
     return  this.abteilung;
   }

}
```

Sichtbarkeit in der Vererbungs-hierarchie

- **`private`**

  Zugriff nur innerhalb der deklarierenden Klasse möglich

- <default>

  Zugriff nur innerhalb des Paketes möglich

- **`protected`**

  Zugriff innerhalb der deklarierenden Klasse möglich

  innerhalb aller Subklassen

  innerhalb des Paketes

- **`public`**

  Zugriff aus allen Klassen heraus möglich

Vererbung und Konstruktoren

- Wenn die Attribute der Basisklasse entsprechend einem Konstruktor der Basisklasse initialisiert werden sollen, muss ein expliziter Aufruf eines Konstruktors der Basisklasse aus einem Konstruktor der Subklasse heraus erfolgen.

  Syntax:

```
super ( parameterliste );
```

- Ein Konstruktoraufruf muss die erste Anweisung im Konstruktor der Subklasse sein.

Vererbung und Polymorphie

- Überschreiben von Methoden und Attributen

  In den abgeleiteten Klassen können Methoden und Attribute der Basisklasse überschrieben/überschattet werden. Das heißt, es werden in der Subklasse Elemente gleichen Namens hinzugefügt.

  Hat eine Methode in einer abgeleiteten Klasse die absolut gleiche Signatur einer Methode der Superklasse, spricht man von überschreiben oder überdecken.

- Polymorphie

  Instanzen der Klasse Person und Mitarbeiter sind von unterschiedlichem Typ. Der Zugriff auf Objekte erfolgt in Java bekannter weise über Referenzen.

```
Person p = new Person( );

Mitarbeiter m = new Mitarbeiter( );
```

Bei Verwendung des Zuweisungsoperators darf einer Referenz einer allgemeinen Basisklasse auch die Referenz einer abgeleiteten Klasse zugewiesen werden, ohne dass der Cast-Operator notwendig wird.

```
p = m; // erlaubt, weil m von p abgeleitet ist
p.vorstellen( );  // welches vorstellen wird hier aufgerufen?
```

Vererbung und der Cast-Operator

- Durch den Effekt des Späten Bindens wird deutlich, dass eine Referenz zur Laufzeit sozusagen die Information mitführt, auf welchen Datentyp es gerade verweist. Dies kann auch programmtechnisch verwendet werden, um zur Laufzeit Objekte zu analysieren. Man „fragt“ eine Referenz, auf welchen Typ sie zeigt.

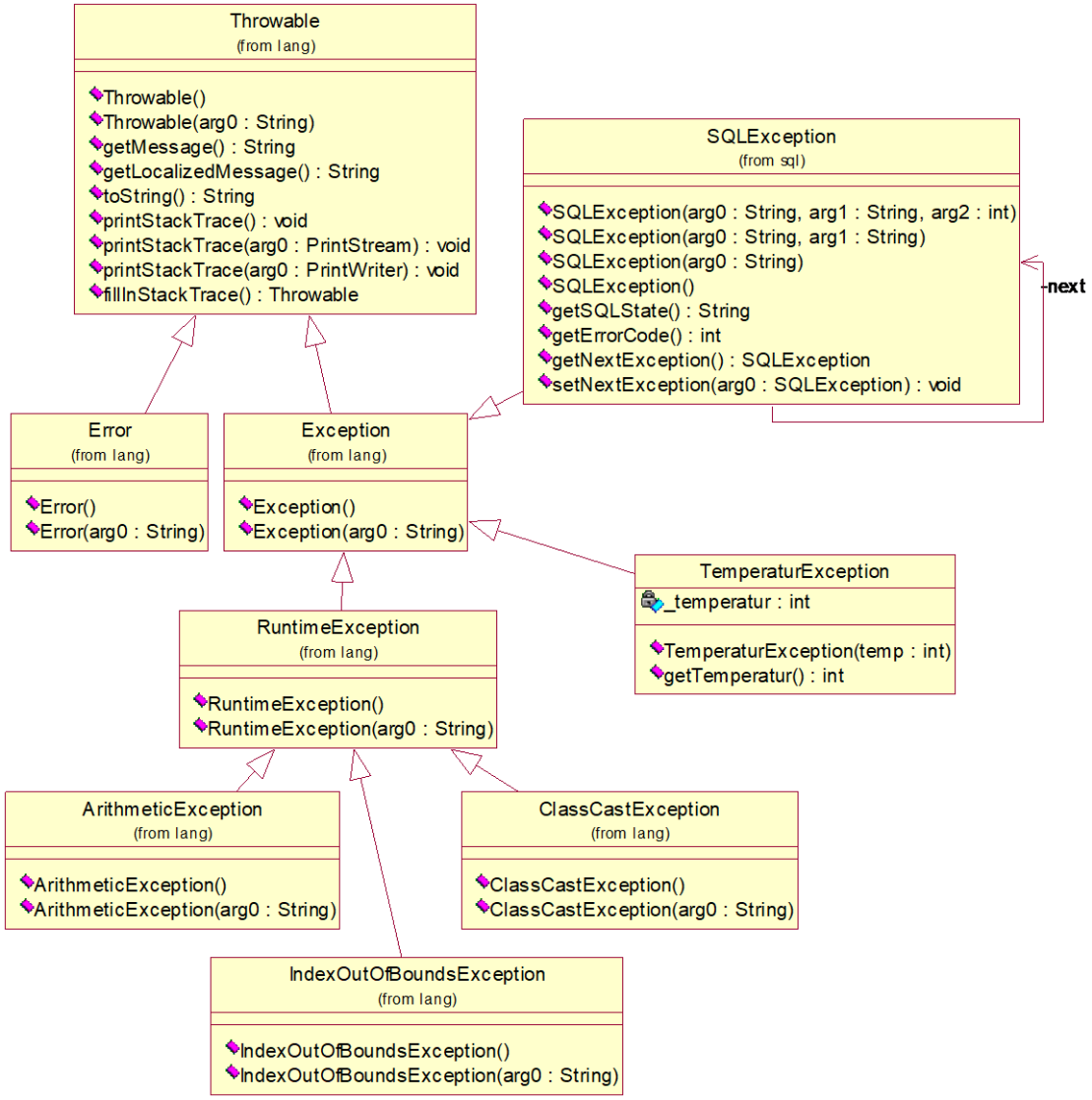
Syntax:

Referenz instanceof Klasse

- Ein Cast von der Basisklasse in die Subklasse wird vom Compiler akzeptiert.

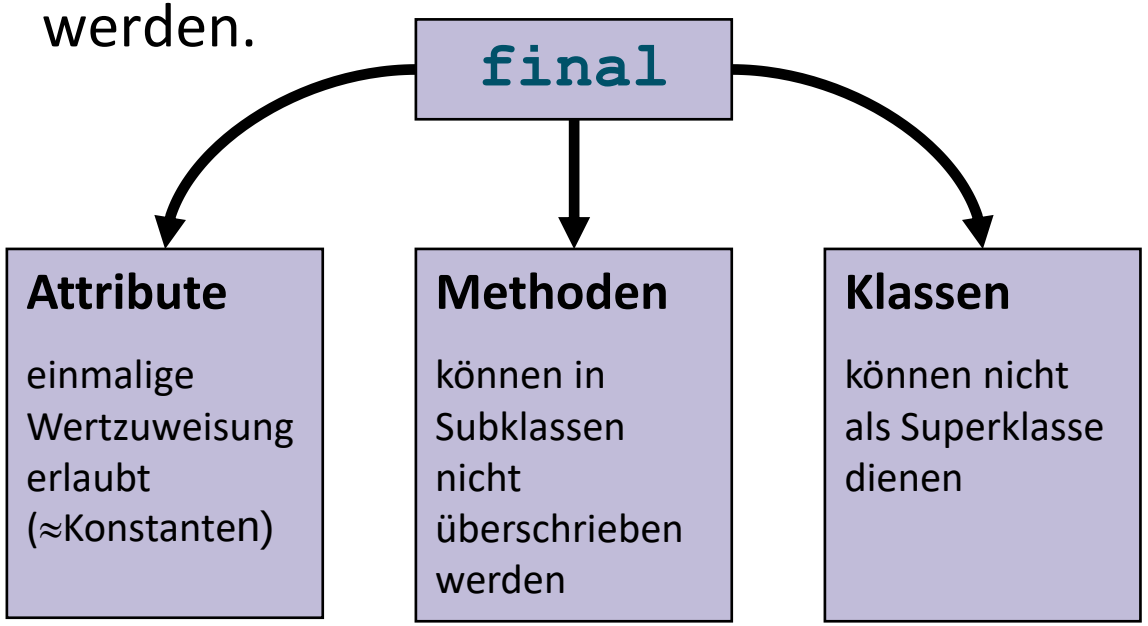
```
public void einmieten( Person p )
{
    if( p instanceof Mitarbeiter )
       Mitarbeiter a = ( Mitarbeiter ) p;  // cast
    //  ...
}
```

Vererbungs-hierarchie der Exceptionklassen



Finale Elemente

- Finale Attribute entsprechen Variablen, die nicht geändert werden können.
- Finale Methoden können in einer Subklasse nicht mehr überschrieben werden.
- Von finalen Klassen kann nicht mehr weiter abgeleitet werden.

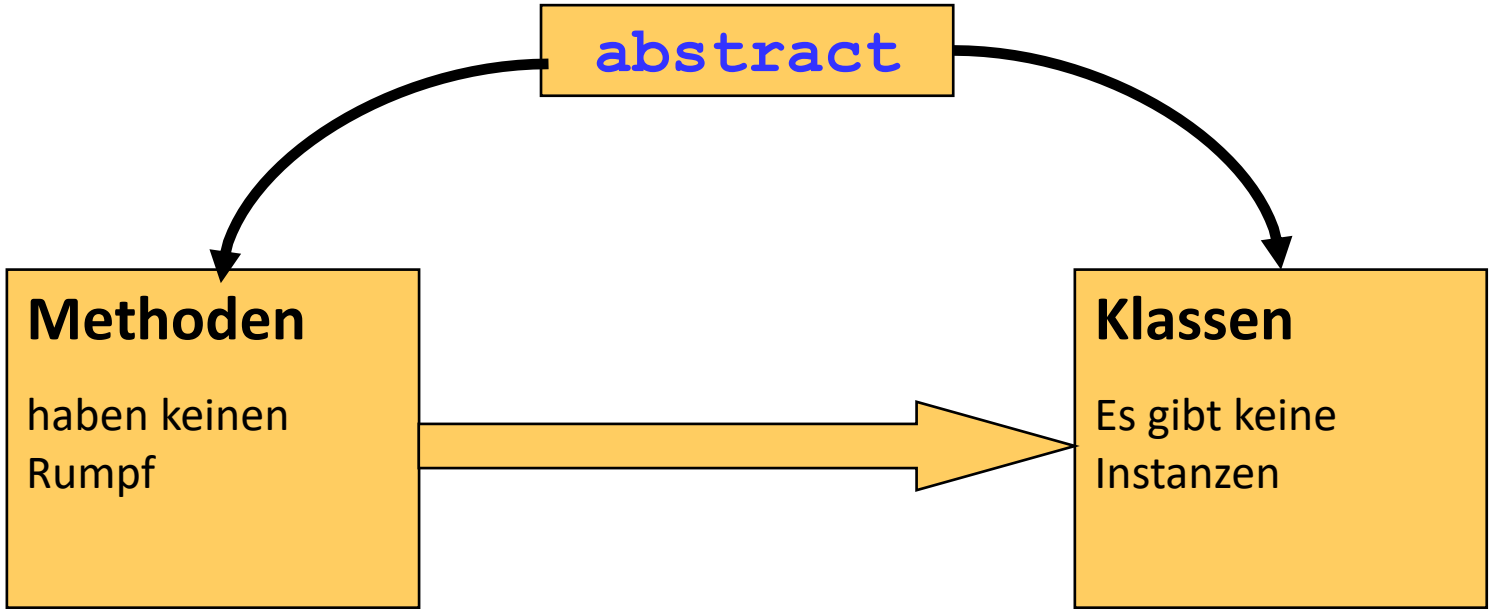


Kapitel 06: Abstrakte Klassen, Interfaces und Pakete

- Abstrakte Klassen
- Interfaces
- Implementierung von Interfaces
- Übersicht der Deklarationen: Klassen
- Übersicht der Deklarationen: Attribute
- Übersicht der Deklarationen: Methoden
- Übersicht der Deklarationen: Konstruktoren
- Pakete
- Statische Importe
- Zugriffsrechte

Abstrakte Klassen

- Von abstrakten Klassen können keine Instanzen erzeugt werden.
- Enthält eine Klasse mindestens eine abstrakte Methode, muss die Klasse selbst abstrakt sein.



Abstrakte Klassen

- Klassendeklaration

```
sichtbarkeit abstract class Klassenname
{
    // Klassenkörper mit Attributen
    // und Methodendeklarationen und -definitionen
}
```

- Methodendeklaration

```
sichtbarkeit abstract typ methodenname( parameter );
  // kein Methodenblock
```

Abstrakte Methoden müssen in abgeleiteten Klassen implementiert werden, sonst ist die Subklasse ebenfalls abstrakt.

Interfaces

- Ein Interface ist per Definition eine Sammlung abstrakter Methoden und statischer, finaler Attribute, auf den ersten Blick also der abstrakten Klasse sehr ähnlich. Der Unterschied liegt darin begründet, dass keine Instanzattribute erlaubt sind.
- Erweiterungen seit JDK8: Interfaces können statische Methoden definieren und Default-Methoden implementieren

Syntax:

```
interface Interfacename {

    // Deklaration statische, finale Attribute
    // und abstrakte Methoden

    // statische Methoden

    // und default Methoden

}
```

Implementierung von Interfaces

- Wird eine Klasse von einem Interface abgeleitet, so muss anstelle des Schlüsselwortes extends das Schlüsselwort implements verwendet werden.
- Für Schnittstellen wird die Mehrfachvererbung unterstützt.

Syntax:

```
sichtbarkeit class KlassenName implements Interface1,
                                          Interface2

{

    // Klassendefinition

}
```

- Eine Klasse, die eine Schnittstelle implementiert, ist solange abstrakt, bis alle im Interface deklarierten abstrakten Methoden definiert wurden.

Übersicht der Deklarationen: Klassen

- Die mit eckigen Klammern umschlossenen Angaben sind optional.

Syntax der Klassendeklaration:

```
[sichtbarkeit] [final] [abstract] class Klassenname
                    [extends Oberklasse]
                    [implements Interface1, Interface2 ...]
{
  // Klassenkörper
}
```

- Sichtbarkeit:

public	Sind auch außerhalb des Paketes sichtbar
<default>	Sind nur innerhalb des Paketes sichtbar

Übersicht der Deklarationen: Attribute

- Syntax der Attributdefinition

```
[sichtbarkeit] [final] [gültigkeit] typ attributname;
```

- Sichtbarkeit

`public`	Sind auch außerhalb der Klasse sichtbar
`protected`	Sind innerhalb der Klasse, allen Klassen der Klassenhierarchie und des selben Paketes sichtbar
`<default>`	Sind innerhalb der Klasse und desselben Paketes sichtbar
`private`	Sind nur innerhalb der Klasse sichtbar

- Gültigkeit

`static`	Klassenattribut
`<default>`	Instanzattribut

Übersicht der Deklarationen: Methoden

- Syntax der Methodendefinition

```
[sichtbarkeit] [final] [abstract] [gültigkeit]
               typ methodenname(  parameterliste )
{
  // Methodenkörper
}
```

- Sichtbarkeit

`public`	`protected`
`<default>`	`private`

- Gültigkeit

`static`	`<default>`

Übersicht der Deklarationen: Konstruktoren

- Syntax der Konstruktordefinition:

```
[sichtbarkeit] Klassenname( parameterliste )
{
  // Aufruf von this( ) oder super( )
  // Konstruktorkörper
}
```

- Sichtbarkeit
  - **public** in allen Klassen sichtbar
  - **protected** innerhalb der Klasse, in allen Subklassen und in allen Klassen desselben Pakets sichtbar
  - **<default>** innerhalb der Klasse und in allen Klassen desselben Paketes sichtbar
  - **private** nur innerhalb der Klasse sichtbar

Pakete

- Pakete (Packages) bündeln Gruppen von Klassen

```
java.lang
java.io
java.awt
de.integrata.grundlagen.oop
```

- Klassen können auf zweierlei Arten angesprochen werden:
  - über vollqualifizierten Klassennamen

```
de.integrata.grundlagen.oop.personen.Person
```

  - über Importieren und mit kurzen Klassennamen ohne Paketnamen
    - nur eine Klasse importieren

```
import de.integrata.grundlagen.oop.personen.Person;
```

    - oder alle Klassen eines Paketes importieren (Achtung: **nicht rekursiv!)**

```
import de.integrata.java.grundlagen.oop.personen.*;
```

Statische Importe

- Import für direkten Zugriff auf statische Attribute und Methoden **`import static`**
- Beispiel: Zugriff auf Konstante **`PI`** und auf Methode **`pow( )`** der Klasse `java.lang.Math`
- Beispiel:

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


public class CircleUtil {

   public static void main( String[ ] args ) {

      CircleUtil util = new CircleUtil( );
      System.out.println( "Area: " + util.area(2.5) );

   }


   public double area( double radius ) {
      return pow( radius, 2d ) * PI;
   }

}
```

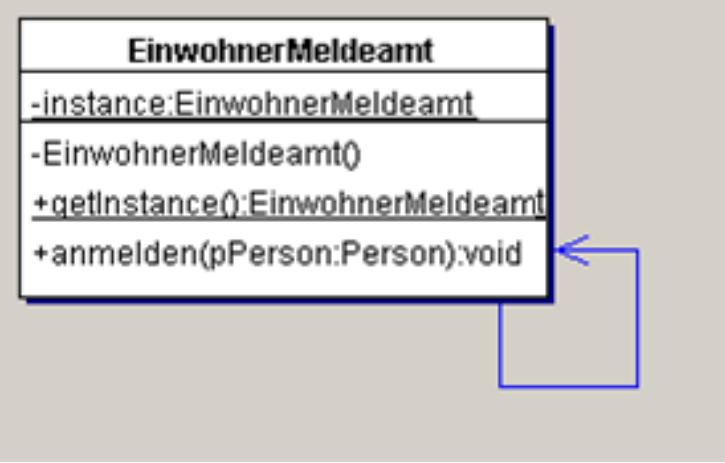
Zugriffsrechte

- **`public`**
  - Zugriff aus allen Klassen heraus möglich
- **`protected`**
  - Zugriff innerhalb der deklarierenden Klasse möglich
  - und innerhalb aller Subklassen möglich
  - und innerhalb des Paketes möglich
- **`<default>`**
  - Zugriff innerhalb der deklarierenden Klasse möglich
  - und innerhalb des Paketes möglich
- **`private`**
  - Zugriff nur innerhalb der deklarierenden Klasse möglich

Kapitel 07: Weiterführende Themen

- Singleton (Design Pattern)
- Assertions
- Verwendung der Assertions
- Wrapper-Klassen
- Enumeration

Singleton (Design Pattern)



- Von einer Klasse soll maximal eine Instanz existieren.
- Eine Klasse mit ausschließlich privaten Konstruktoren.
- Die Klasse enthält ein privates statisches Attribut von sich selbst.
- Die Klasse definiert eine öffentliche statische Zugriffsmethode auf dieses Attribut.
- Die Instanzierung des Objektes erfolgt exakt ein Mal, geprüft in der statischen Zugriffsmethode.

Assertions

- Schüsselwort `assert`
  - **`assert <boolescher Ausdruck>`**
  - Ausdruck **`true`**: Fortführung des Programmlaufs
  - Ausdruck **`false`**: Abbruch mit AssertionError
- Assertions sollen Programmierfehler der Anwendung signalisieren ("darf nie auftreten").
  - Wenn eine Assertion nicht erfüllt ist, ist das Programm nicht mehr sinnvoll weiter zu führen.
- Gegensatz: Exceptions für Reaktionen auf Bedienungsfehler.
- Assertions können zur Laufzeit aktiviert oder deaktiviert werden
  - **`-enableassertions`** oder kurz **`-ea`**
  - **`-disableassertions`**, oder kurz **`-da`**
- Steuerung bis auf Klassenebene möglich
  - **`java -ea:de.integrata.java.sprache.assertions.AssertDemo`**

Verwendung der Assertions

- Vorbedingungen prüfen
  - Assertion nur sinnvoll für private Methoden: Der Entwickler der Klasse verwendet eine interne Methode falsch
  - Bei öffentlichen Methoden: Exceptions für Reaktion auf externe Fehlbenutzung
- Nachbedingungen prüfen
  - Assertion: Unvorhergesehener Programmlauf führt zu inkonsistentem Objektzustand.
  - Exception: Fehlerhafte Benutzung wird dem Aufrufer mitgeteilt.

Wrapper-Klassen

- Zu einfachen Datentypen entsprechende Wrapper-Klassen:

```
byte        Byte
short       Short
int         Integer
long        Long
char        Character
float       Float
double      Double
boolean     Boolean
```

- Eigenschaften und Verwendung
  - enthalten den einfachen Typ als Attribut
  - stellen Konvertierungsmethoden zur Verfügung
  - ermöglichen call-by-reference für die einfachen Datentypen
- Seit der Sprachversion 5.0 konvertiert der Compiler automatisch zwischen den einfachen Datentypen und den Wrapper-Klassen
  - „Autoboxing/Unboxing“

Enumeration

Beispiel:

- Deklaration

```
public enum Season
{
    SPRING, SUMMER, FALL, WINTER;
   // und Attributdeklarationen und Methodendefinitionen
}
```

- Jede Enumeration ermöglicht den Zugriff auf ihre Werte durch die Methode values(). Die Iteration erfolgt sehr einfach innerhalb einer for-Schleife:

```
// values( ) liefert ein Array mit allen Referenzen
Season[ ] allRefs = Season.values( );
for ( int i = 0; i < allRefs.length; i++ ) {
      System.out.println( allRefs[ i ] );
}
```

Kapitel 08: Klassen der Klassenbibliothek

- Die Klasse String
- Die Klassen StringBuilder und StringBuffer
- Die Klasse Object
- Object: clone()
  Flache und tiefe Kopien
- Einige Klassen des Paketes java.util
- Konfiguration mit System-Properties
- Formatierte Ausgaben
- Weitere Klassen

Die Klasse String

- Eine Abfolge von Buchstaben, die der Unicode-Kodierung entsprechen
  - Implementiert durch die String-Klasse
  - Enthalten in der Standard-Java-Bibliothek
  - Objekte vom Typ String sind konstant und nach der Instanzierung nicht veränderbar (immutable)

```
String arg;

arg = "Die Eingabe enthielt";
```

- Zeichenketten können mit dem Operator "+" zusammengefügt werden

```
arg = arg + " Buchstaben";
```

  - Alternativ kann die Methode `concat( )` verwendet werden.
- Die Klasse String unterstützt unter anderem

`length()`	Abfragen der Länge
`charAt(int index)`	Abfragen von einzelnen Zeichen
`equals(String s)`	Vergleich von Strings

u.v.a.m.

Die Klassen StringBuilder und StringBuffer

- Die Klasse **`StringBuffer`** dient zur Zeichenkettenverarbeitung
  - Als Erweiterung zu String wird hier bei der Instanzierung ein zusätzlicher Bufferbereich besorgt
    - `new StringBuilder(int länge)` // Puffer erzeugen
    - `setLength(int länge)` // Länge neu setzen
    - `append(...)` // Zeichen oder Zeichenketten anhängen
    - `insert(int wo, String text)` // Text einfügen
- Seit Java 5 wird StringBuffer durch die Klasse StringBuilder ergänzt
  - Kompatibel zu **`StringBuffer`**, aber nicht synchronisiert
  - Ersatz für **`StringBuffer`** in einer Single-Thread-Umgebung.
  - Instanzen von **`StringBuilder`** sind unsicher (nicht Thread-Safe) in Umgebungen mit mehreren Threads

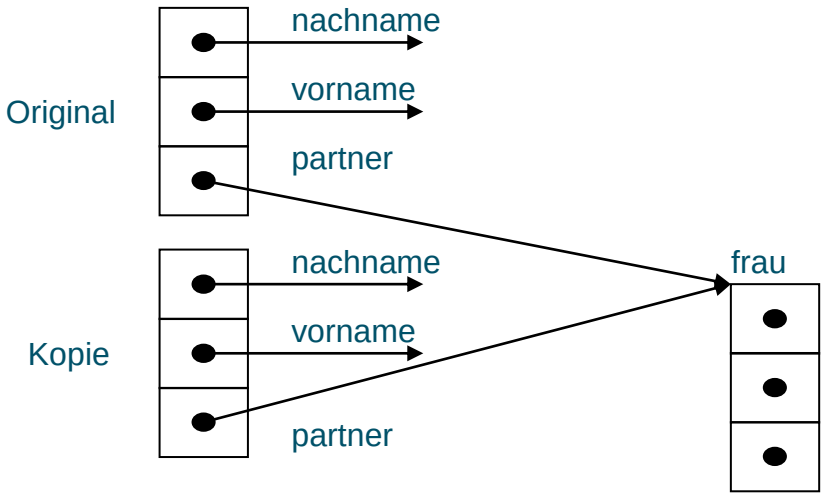
Die Klasse Object

- Jede Klasse ist direkt oder indirekt von der Klasse `java.lang.Object` abgeleitet

- Object deklariert verschiedene allgemeine Methoden

  - `boolean equals( Object )` // Vergleicht zwei Objekte miteinander
  - `String toString( )` // Zeichenkettenrepräsentation
  - `Object clone()` // Erzeugt eine Kopie

Object: clone() Flache und tiefe Kopien



- Flache Kopie
  - In der Kopie stehen die kopierten Werte
  - Bei einfachen Datentypen ist dies eine reale Kopie
  - Wenn Referenzen auf Objekte oder Arrays kopiert werden, zeigt die Kopie auf das gleiche Objekt!
  - Wenn Objekte kopiert /"gecloned" werden, wird eine bitweise Kopie des Objektspeichers angefertigt.
    - Vorhandene Referenzen zeigen auf das selbe Objekt.
- Tiefe Kopie
  - Auch referenzierte Objekte und Arrays werden kopiert
  - Dies liegt in der Verantwortung des Programmierers
  - Überschreiben und Nutzen der **`clone()`**-Methode aus Object.

Einige Klassen des Paketes java.util

- `Calendar, Date`
  - Datum als Klasse
- `Vector, ArrayList`
  - Dynamischer Container für beliebige Objektreferenzen
- `Enumeration, Iterator`
  - Aufzählung von Elementen
- `Hashtable, HashMap`
  - Assoziatives Array

Konfiguration mit System-Properties

- Neben den Aufrufparametern der main-Methode können auch System-Properties gesetzt und gelesen werden
  - Auslesen durch
    ```
    System.getProperty( String propertyName )
    ```
  - Setzen durch Aufruf mit der Option „-D“
    ```
    java -Dkey=value …
    ```
- Hinweis: Auch eigene Properties können verwendet werden
  - `java.util.Properties` Lesen/Schreiben aus Dateien
  - `java.util.prefs.Preferences` zum Lesen benutzerabhängiger Eigenschaften

Formatierte Ausgaben

- Dazu dient das Paket **`java.text`**
  - **`NumberFormat`**, **`DecimalFormat`**
  - **`DateFormat`**, **`SimpleDateFormat`**
- Länderspezifische Einstellungen durch vordefinierte Konstanten
  - **`Locale`**

```
import java.text.NumberFormat;
import java.util.Locale;

public class App {
  public static void main( String[ ] args ) {
    // get format for default locale
    NumberFormat nf1 = NumberFormat.getInstance( );
    System.out.println( nf1.format( 1234.56 ) );
    // get format for German locale
    NumberFormat nf2 = NumberFormat.getInstance( Locale.GERMAN );
    System.out.println( nf2.format( 1234.56 ) );
  }
}
```

Weitere Klassen

- Das JDK enthält die Dokumentation aller Klassen und Interfaces im HTML-Format
  - auf

    http://download.oracle.com/javase/7/docs/api/
    http://download.oracle.com/javase/8/docs/api/
    http://download.oracle.com/javase/9/docs/api/

  - oder lokal installiert

    ${Lokales-API-Installationsdirectory}\api\index.html

Kapitel 09: Literatur

- Literatur und Webseiten
- Literatur UML

Literatur und Webseiten

englisch:

- Online-Dokumentation (API) innerhalb des JDK
  http://download.oracle.com/javase/9/docs/api/
- http://www.oracle.com/technetwork/java/index.html
- http://download.oracle.com/javase/tutorial/index.html
- Newsgruppen
  http://groups.google.com/group/de.comp.lang.java/topics

deutsch:

- http://www.javabuch.de/ Java-Programmierung
- http://openbook.rheinwerk-verlag.de/javainsel/ Java ist auch eine Insel
- http://openbook.rheinwerk-verlag.de/java7/ Java – mehr als eine Insel
- Forum
  http://www.java-forum.org/
- http://www.tutorials.de/java/

Literatur UML

- UML ist ein Standard der OMG (http://www.omg.org/uml) und definiert eine Notation zur Visualisierung, Konstruktion und Dokumentation von Modellen für die objektorientierte Softwareentwicklung.

- Literatur:

[Booch 94] Grady Booch: Object oriented design with applications.

[Booch 96] Grady Booch: Object Solutions. Managing the Object-Oriented Projekt. Addison Wesley 1996Booch, Rumbaugh, Jacobson: The Unified Modeling Language User Guide, Addison Wesley 1998

Booch, Rumbaugh, Jacobson: The Unified Modeling Language Reference Manual, Addison Wesley 1998Booch, Rumbaugh, Jacobson: The Unified Software Development Process, Addison Wesley 1999

[Coad 93] Peter Coad / Jill Nicola: Object-oriented programming. Yourdon-Press 1993

[Gamma 92] Erich Gamma: Objektorientierte Software-Entwicklung am Beispiel von ET++. Springer 1992

[Gamma 94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns: Elements of Object-Oriented Software Architecture. Addison Wesley 1994

Jacobson et.al.: Object-oriented Software engineering, Addison Wesley 1992

[Meyer 88] Bertrand Meyer: Object-oriented Software Construction. Prentice Hall 1988

James Rumbaugh et. al: Object-oriented modeling and design (OMT), Prentice Hall, 1994

[WWW 90] Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener: Objektorientiertes Software-Design. Prentice Hall 1990, in Deutsch beim Carl Hanser Verlag.

Kapitel 10: Anhang

- Schlüsselwörter
- Tool: javadoc

Schlüsselwörter

- Folgende alphabetische Auflistung enthält die reservierten Java Wörter:

abstract assert boolean break byte case catch char class continue const default do double else enum extends final finally float for goto if implements import int instanceof interface long new native package private protected public return short switch static super strictfp synchronized this throw throws transient try void volatile while

- Nicht verwendet, nur reserviert:

const goto

Tool: javadoc

- `javadoc` erzeugt Programmdokumentation (HTML-Seiten, Aufbau und Format wie API-Dokumentation)
- durchsucht Quelldateien nach Deklarationen und `javadoc`-Kommentaren (`/**...*/`)
  - Kommentar = Text im HTML-Format
  - spezielle Tags:
    - @see Klasse
    - @see Klasse#Methode
    - @version Versionsname oder -nummer
    - @author Name des Autors
    - @return Beschreibung des Rückgabewertes
    - @throws Exceptionbeschreibung
    - @param Parameterbeschreibung

Tool: javadoc

- Beispiele für den Aufruf von `javadoc`
  - **`javadoc *.java`**
    - Alle Quellcodes werden im aktuellen Verzeichnis dokumentiert (nur die öffentliche Schnittstelle)
  - **`javadoc -private -d doc *.java`**
    - Alle Quellcodes inklusive der privaten Elemente werden im Unterverzeichnis doc dokumentiert
- Oder mittels der Entwicklungsumgebung Eclipse

Copyright und Impressum

Integrata AG
Zettachring 4
70567 Stuttgart

integrata
Cegos Group
inspire
qualify
change