Zusammenfassung Objektorientierte Programmierung

# Grundlagen Programmieren / Java

Programmieren ist das Lösen von Problemen mittels Computer:

1. Anforderung: Was brauche ich, was sind die Probleme
2. Entwurf: Lösungsweg definieren, wie will ich vorgehen
3. Implementierung: Programm schreiben
4. Testen und verbessern
5. Evolution: Betrieb Änderungsbedarf erkennen --> Optimierungen

Programmiersprache

* Algorithmen und Datenstrukturen
* Syntax quasi Grammatik-> Struktur der Sprachelemente(strukturelle Organisation)
* Semantik(Bedeutung der Sprachelemente und dessen Zusammenspiel --> beispielsweise Expression muss ein «boolean» verlangen --> true / false --> Eindeutigkeit

Algorithmus

Algorithmus: Definiert in welcher Reihenfolge das Programm Schritte ausführen muss

Anfangszustand

Abfolge von definierten Operationen

Endzustand --> Resultat

Deklaration / Initialisierung

Programmcode 
public class Factorial { 
public static void main(String[] args) { 
int n; 
int p; 
int i; 
n 
i 
- 12; 
1; 
while (i <= n) 
i; 
i 
Deklarationen 
Initialisierungen 
Kern des Algorithmus 
System. out. print In (p); 
Ausgabe 
HSR 
TECHNIK 
FHO Fachhochschule Ostschweiz 
OO HS2017 - 
Einführung 
13 

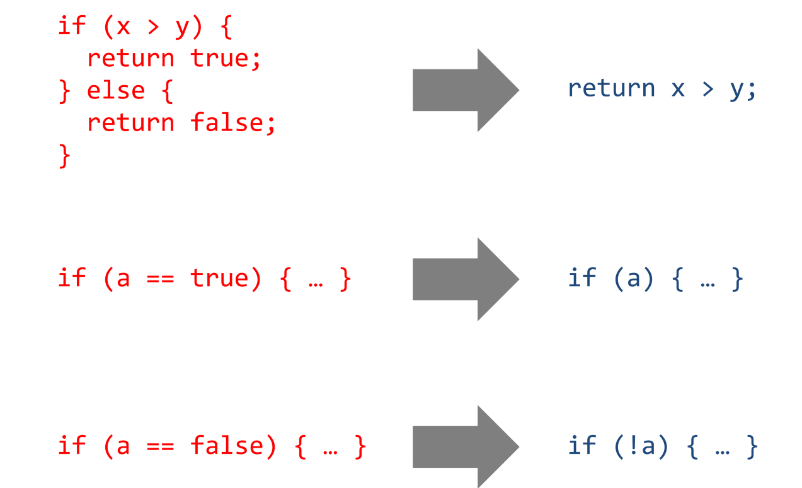
Kompilieren

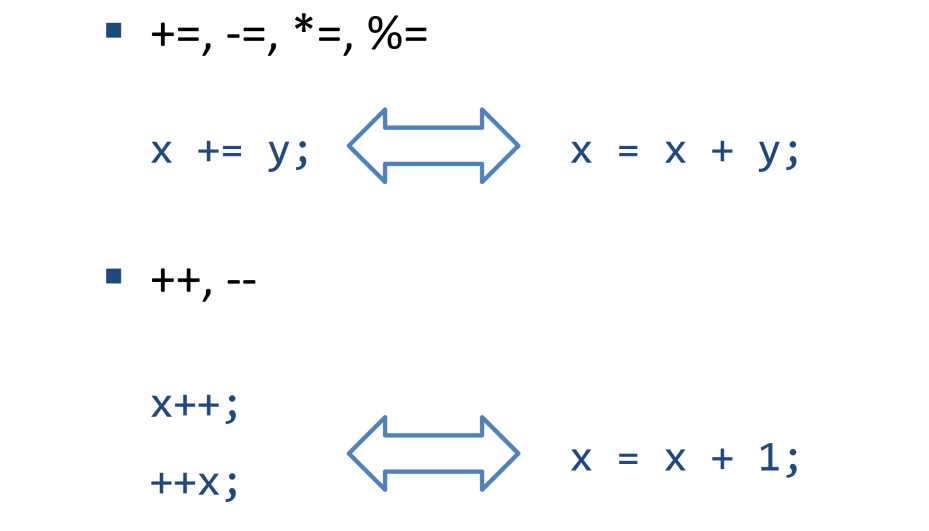
Kompilieren ist das Umwandeln von Programmcode zu ausführbarem Code. In Java übernimmt diese Aufgabe die Java Compiler

Namenskonventionen

* **Variablen, Methoden**
  + **Beginnt mit Kleinbuchstabe**
  + **Worttrennung mit Grossbuchstabe ab zweitem Wort**
  + **int inputValue**
  + **int readNumber() {}**
* **Klassen**
  + **Erster Buchstabe Grossbuchstabe, Rest Kleinbuchstaben**
  + **Worttrennung Neuer Grossbuchstabe**
  + **Vehicles**
  + **ScientificCalculator**
* **Allgemein**
  + **Treffende Namen**
  + **Bevorzugt Englisch**
  + **Keine Abkürzungen**
  + **Keine Namen wie: msg4you, cpyobj oder MyClass, etc.**

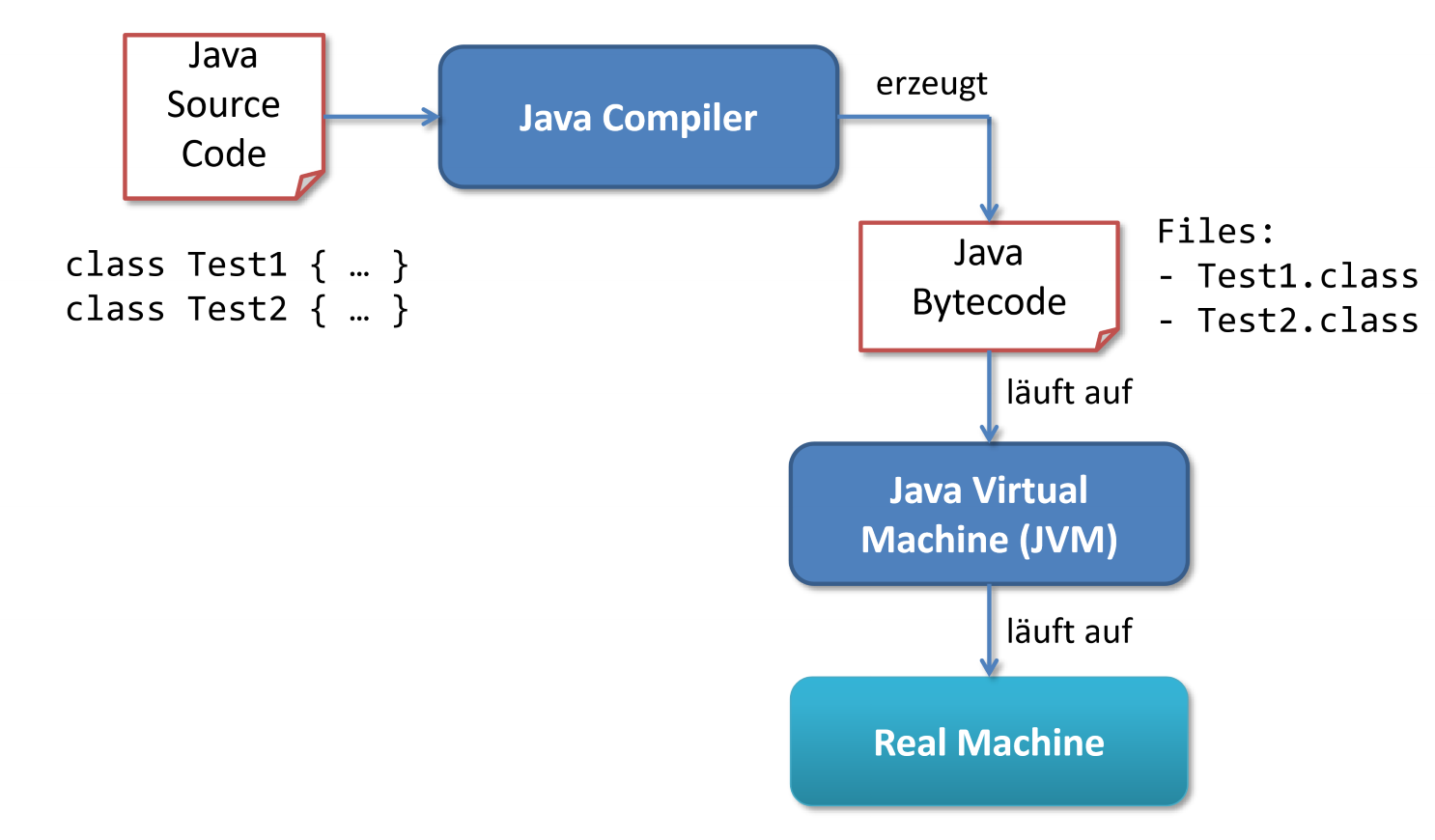
Stilverbesserungen







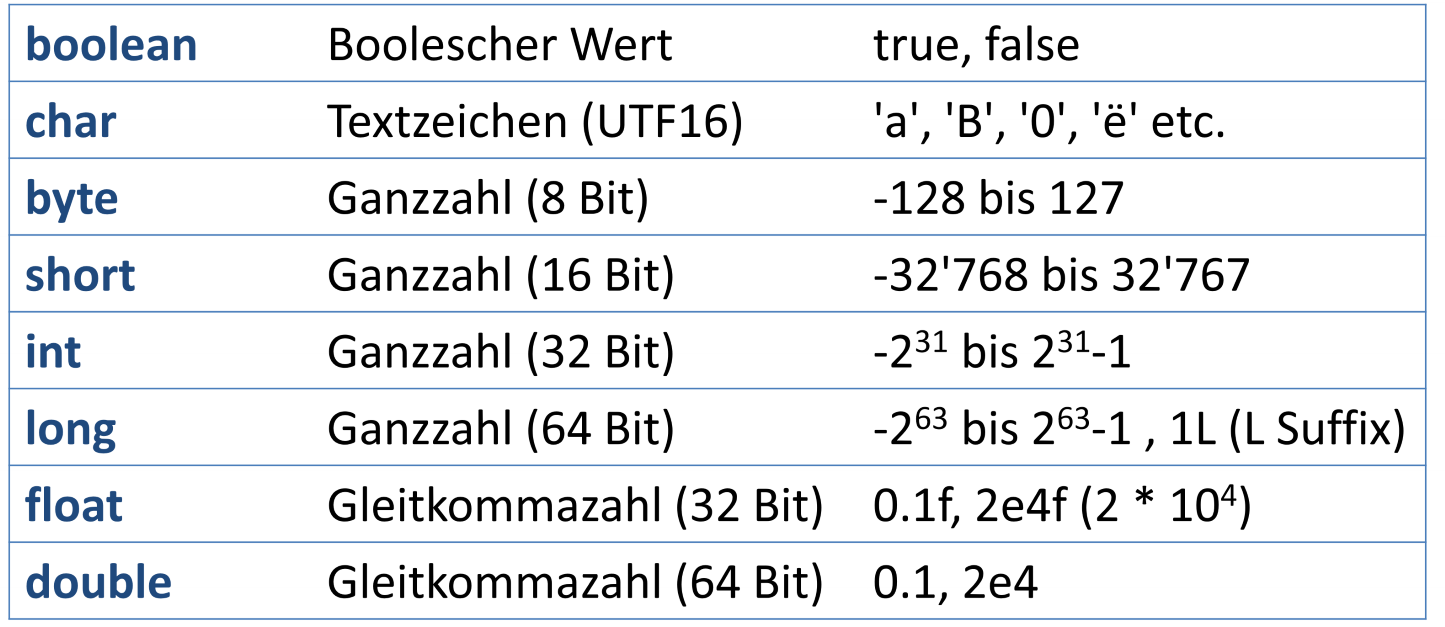
Java Laufzeit Architektur

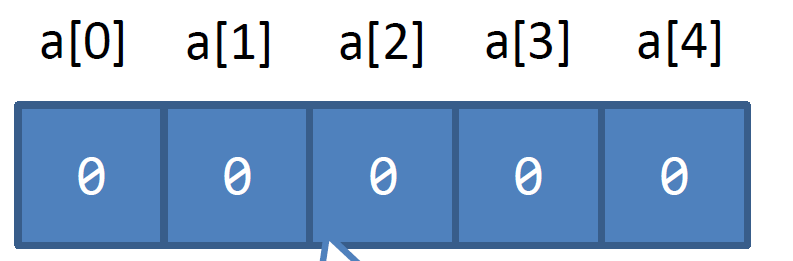


# Datentypen / Operationen / Kontrollstrukturen

Primitive Datentypen

Beachte: Alle primitiven Datentypen sind kleingeschrieben und vorzeichenbehaftet(signed):



Arrays

**int**[] **intArray** = **new** **int**[4]; //fixe Grösse definieren

**int**[] **intArray2** = {1, 2, 3}; //Direkt Initialisieren, Grösse festgelegt

**System**.***out***.println(intArray.length); //Ausgabe Länge Array hier 4

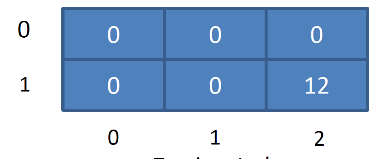
**System**.***out***.println(intArray2.length); //Ausgabe Länge Array hier 3

**for**(**int** **i** = 0; i<intArray.length; i++) //Länge des Arrays kann in For-Schleife

{

**System**.***out***.println(intArray[i]);

}



Mehrdimensionale Arrays

**int**[][] **intArray** = **new** **int**[2][3]; //fixe Grösse definieren [Zeilen][Spalten]

**int**[][] **intArray2** = {  
 {1, 2, 3}, //Erste Zeile füllen

{4, 5, 6} //Zweite Zeile füllen

}; //Direkt Initialisieren, Grösse festgelegt

**System**.***out***.println(intArray.length); //Ausgabe Anzahl Zeilen = 2 **System**.***out***.println(intArray[0].length); //Ausgabe Anzahl Spalen = 3;

**System**.***out***.println(intArray2.length); //Ausgabe Anzahl Zeilen = 2 **System**.***out***.println(intArray2[0].length); //Ausgabe Anzahl Spalen = 3;

Arithmetische Operatoren

Wie in der Mathematik:

1. Klammern
2. Multiplikation / Division / Modulo
3. Addition Subtraktion

**System**.***out***.println((1 + 3 % 2) / 2 - 1 \* 3 + 5 );

1. 3 % 2 = 1
2. 1 + 1 = 2
3. 2 / 2 = 1
4. 1 \* 3 = 3
5. 1 – 3 = -2
6. -2 + 5 = 3

Unäre Operatoren ++ & --

**int** **x** = 1;

**System**.***out***.println(x++); // = 1, zuerst ausgelsen, dann inkrementieren

**System**.***out***.println(++x);// = 3, zuerst inkrementieren, dann ausgelesen

Gleiches für x—und --x

Aufpassen:

**int** **x** = 1;

**int** **y** = (x++)+(++x);

**System**.***out***.println("" +x +" " +y); //x = 3, y= 4 Klammern beachten!

Switch-Case

Um genaue Werte abzufragen, ansonsten if/else verwenden.

**int** **i** = 2;

**switch**(i){

**case** 0:

**System**.***out***.println("i ist null");

**break**;

**case** 1:

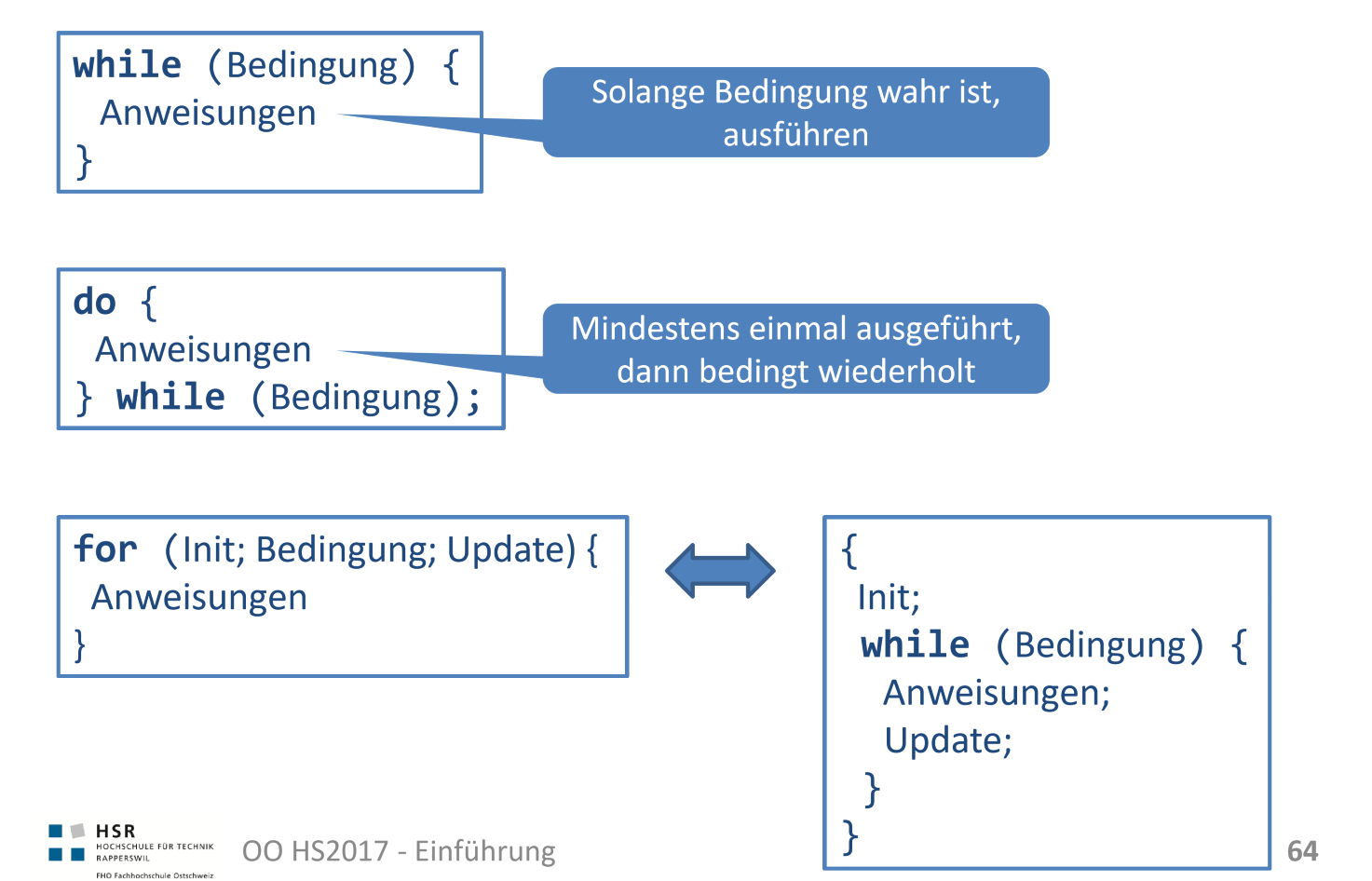
**System**.***out***.println("i ist 1");

**break**;

**default**:

**System**.***out***.println("i ist ungleich 0 und 1");

while, do, for



ForEach

Mit der ForEach-Schleife ist es möglich durch Arrays zu iterieren.

**int**[] **iArray** = {1, 2, 3};

**for** (**int** **i** : iArray)

{

**System**.***out***.println(i);

}

Oder:

**String**[] **stringArray** = {"Hallo", "World"};

**for** (**String** **string** : stringArray)

{

**System**.***out***.println(string);

}

Oder

**String** **string** = "HalloWorld";

**char**[] **charArray** = string.toCharArray();

**for** (**char** **c** : charArray)

{

**System**.***out***.println(c);

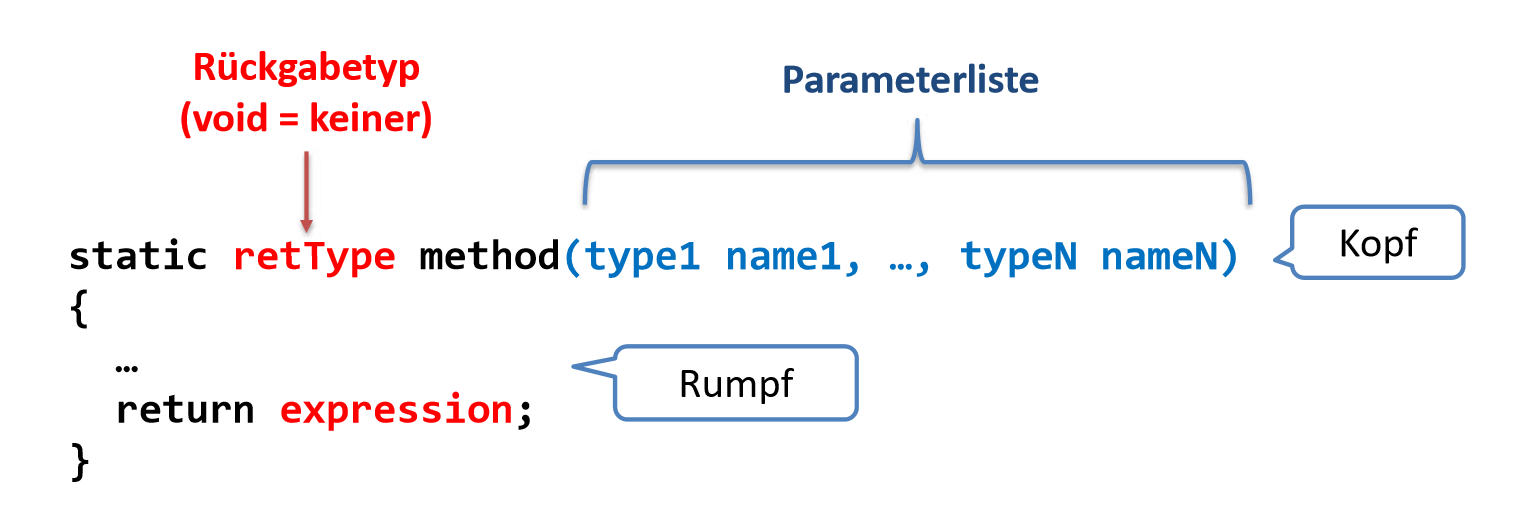
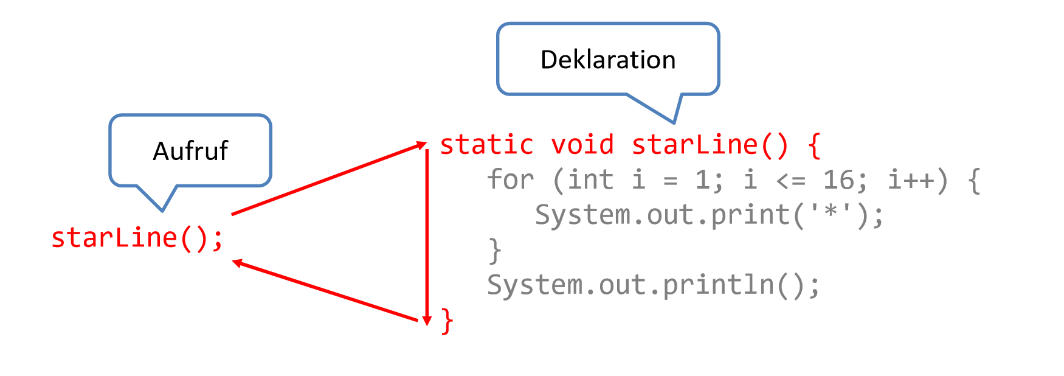
}

Die Ausgebe funktioniert auch, wenn man ein Array aus Objekten hat. Später mehr dazu.

# Methoden und Rekursion

Methoden

**Grundlage**

****Container mit einem Name, in welchem sich Programmcode befindet. Kann über den Namen aufgerufen werden.

**Statische Methode / Variable**

* Gehört keinem Objekt, keine Objektorientierung
* Umgebende Klasse somit dient nur als Gruppierung

**public** **class** **Stars**

{

**static** **int** *stars* = 3;

**public** **static** **void** **main**(**String**[] args)

{

*stars*++;

*printStars*();

}

**private** **static** **void** **printStars**()

{

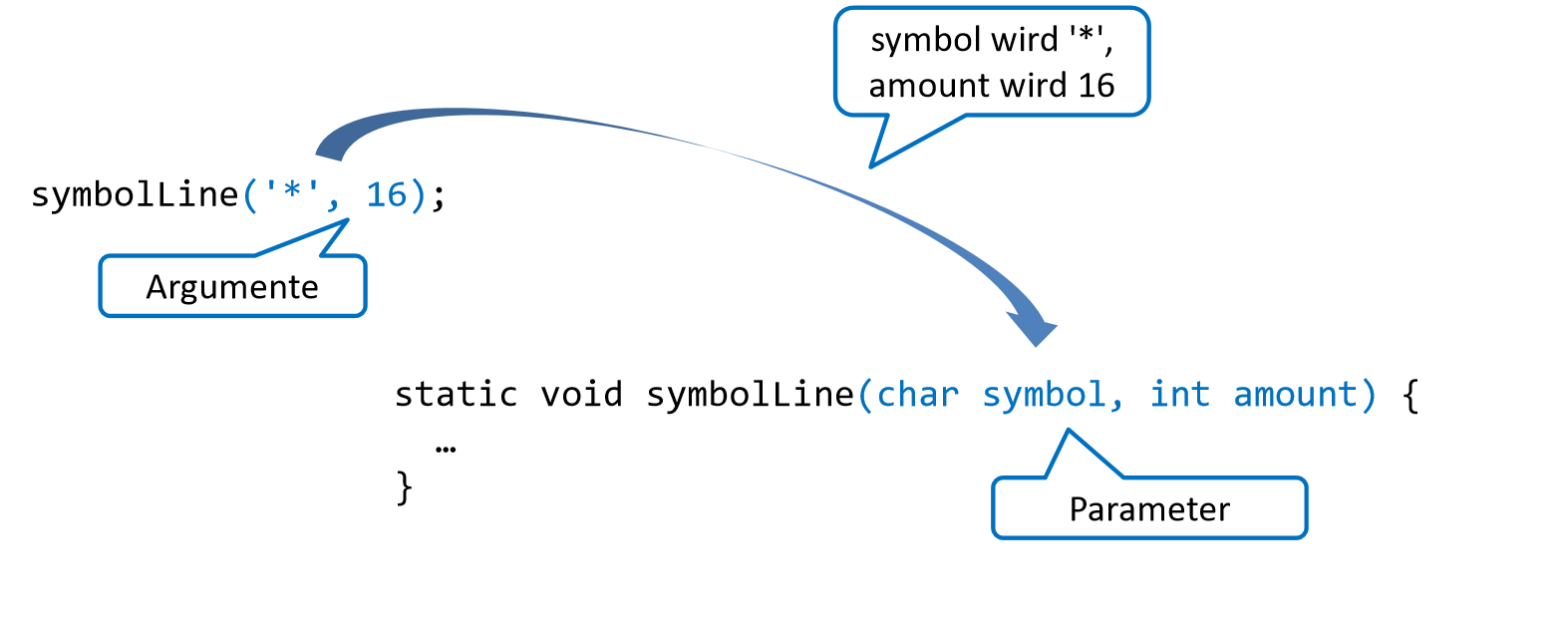
**System**.***out***.println(++*stars*);

}

}

Statische Methoden werden «nur» gebraucht in der Klasse, wo sich auch die Main-Methode befindet. Statische Variablen grundsätzlich vermeiden.

**Parameter / Argumente**

****

**Call by Value**

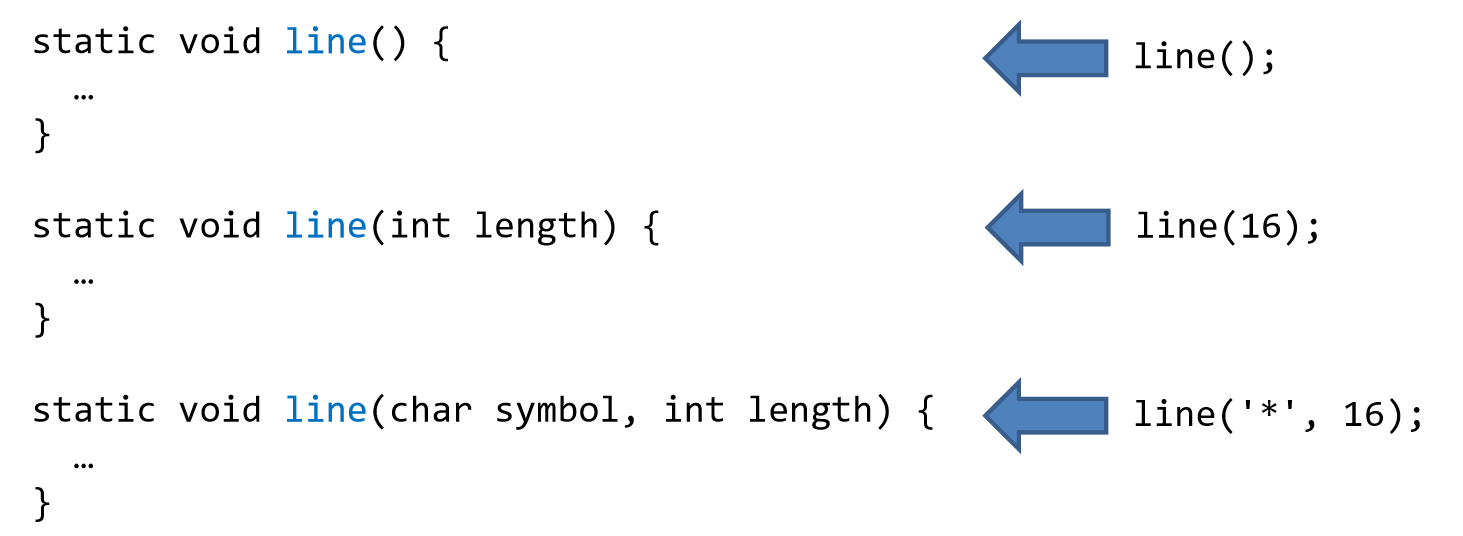
Java unterstützt nur Call by Value**:**

* Eine Kopie wird übergeben
* Änderung an Parameter bei Aufrufer nicht sichtbar

Achtung: Es gilt immer Call by Value, jedoch wird bei einem Array eine Kopie der Adresse übergeben. Das heisst die Adresse ist die gleiche und somit ist es das selbe Array keine Kopie.

**Overloading**

Eine Methode ist eindeutig definiert durch Ihre Signatur. Die Signatur besteht aus dem Methodennamen und den Parametern

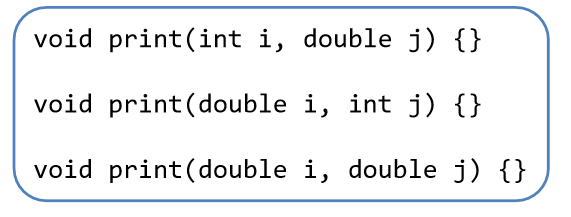
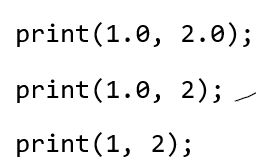
Overloading tritt auf, wenn mehrere Methoden mit gleichem Namen in einer Klasse vorkommen. Diese werden dann anhand der Signatur(Methodenname + Parameter) identifiziert.

Auswahl der Methode zu Compile-Zeit

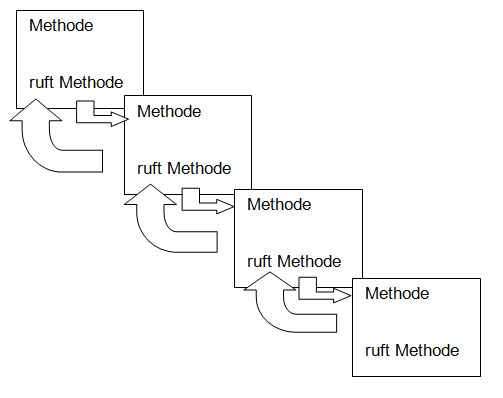
1. Anzahl Parameter entscheidend
2. Argumente müssen auf Parametertyp passen

Wenn mehrere Kandidaten(Oberload Resolution) wird der spezifischere gewählt:

* F ist spezifischer, wenn alle möglichen Aufrufe für F auch für G gehen aber nicht umgekehrt:



1. Geht nur 3. Methode
2. 2. Methode spezifischer
3. Compiler Fehler weil 1 & 2 gleich spezfisch



Rekursion

Rekursion nennt man ein Verfahren, wo sich eine Methode immer wieder selber aufruft:

**private** **static** **boolean** **checkPalindrom**(**char**[] testString, **int** start, **int** end)

{

**if** (start > end)

{

**return** **true**;

} **else**

{

**if** (testString[start] == testString[end])

{

**return** *checkPalindrom*(testString, start + 1, end - 1);

} **else**

{

**return** **false**;

}

}

Es wird rekursiv geprüft, ob ein Palindrom vorliegt.

**public** **static** **void** **printStrings**(**int** length, **String** string)

{

**if** (length <= 0)

{

**System**.***out***.println(string);

} **else**

{

**for** (**char** **c** = 'A'; c <= 'Z'; c++)

{

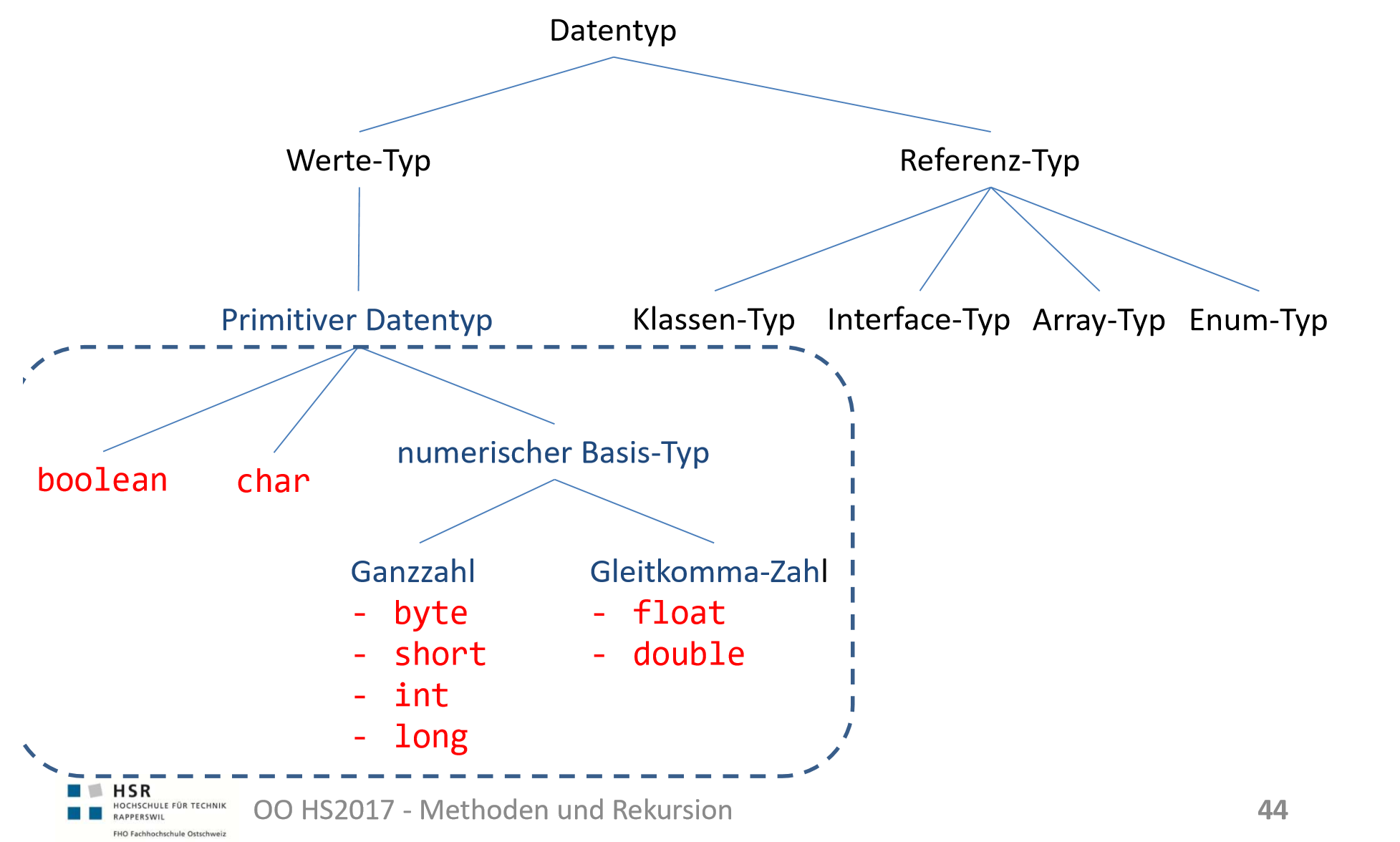
*printStrings*(length - 1, string + c);

}

Es werden rekursiv alle Charakter-Kombinationen von einer bedingten Länge angegeben.(Brutforce)

Andere Beispiele anschauen!!

Datentypen erweitert



Integer Literale

**System**.***out***.println(036); // oktal --> dezimal = 30

**System**.***out***.println(0b10); //biniär --> dezimal = 2

**System**.***out***.println(0xFF); // hexadezimal --> dezimal = 255

* 0<Wert> oktal
* 0b<Wert> binär
* 0x<Wert>

Ganzzahl Literale

**System**.***err***.println(19l);

**System**.***out***.println(0xFFl);

L am Schluss kennzeichnet, dass es sich um ein Long handelt.(l

Gleitkommazahl Literale

**System**.***err***.println(2.3E4f); //= 23000.0

**System**.***out***.println(15E-3d); // =0.015

* F für float
* D für double
* E für ^10
* Ohne Angabe immer Double(64bit)

Numerische Fehler bei Float & Double

Bei Float und Double kann es wegen Rundungsunterschieden zu Fehler kommen. Deshalb Vergleiche vermeiden und mit math.abs(double – long) vergleichen.

**float** **f1** = 1f / 3f; // = 0.33333334

**double** **d1** = 1d / 3d;// = 0.3333333333333333

**System**.***out***.println(f1 == d1); // false wegen Rundung

Überlauf & Division durch 0 beachten

**Überlauf**

* Ganzzahlen 🡪 Falsches Ergebnis
* Gleitkommazahlen 🡪Double.POSITIVE\_INFINITY

**Division durch 0**

* Ganzzahlen 🡪 Exception / by zero(auch bei %)
* Gleitkommazahlen 🡪
  + Double.POSITIVE\_INFINITY
  + Double.NEGATIVE\_INFINITY
* Ausnahmen
  + NaN(Not a Number)
  + Beispiele 0.0 / 0.0, sin(2), etc.

Text Literale

* Character
  + ‘a’, ‘A’
* Escapes
  + ‘\n’ 🡪 neue Linie
  + ‘\’’ 🡪 Apostroph
  + ‘\\’ 🡪 Baslash
  + Etc.

Implizite Typenkonversion

**int** **i1** = 7;

**double** **d** = 3;

**float** **f** = 4;

**System**.***out***.println("int/int(7i/4) "+i1/4); // = 1 (+ Rest 3) Typ int

**System**.***out***.println("int/double(7i/3d) " +i1/d); //= 2.3333 Typ double

**System**.***out***.println("int/float(7i/4f) " +i1/f); //= 1.57 Typ Float

**System**.***out***.println("int/2.3(7i/2.3) " +i1/2.3); //= 3.0434 Typ double

**System**.***out***.println("double / int(3d/2) " +d/2); // = 1.5 Typ double

**System**.***out***.println("double / int(3d/3) " +d/3); // = 1.0 Typ double

**System**.***out***.println("float/int(4f/3) " +f/3); //= 1.333 Typ float

**System**.***out***.println("float/int(4f/2) " +f/2); // = 2.0 Typ float

**System**.***out***.println("int/0(7i/0) " +i1/0); // Exception / by zero

**System**.***out***.println("d/0(3d/0) " +d/0.0); // Infinity

**System**.***out***.println("f/0(f/0) " +f/0.0); // Infinity

**System**.***out***.println(i1/(-4)); // = -1

**System**.***out***.println(1 + 3 / 2); // = 2 int

**System**.***out***.println(1.0 + 3 / 2); // 2.0 double

**System**.***out***.println(1 + 3.0 / 2); // = 2.5 double

**System**.***out***.println(1 + 3 / 0); // Exception / by Zero

**System**.***out***.println(1 + 3.0 / 0); //Infinity

**System**.***out***.println(1 + 3.0 / 0.0); // Infinity

Explizite Typenkonversion

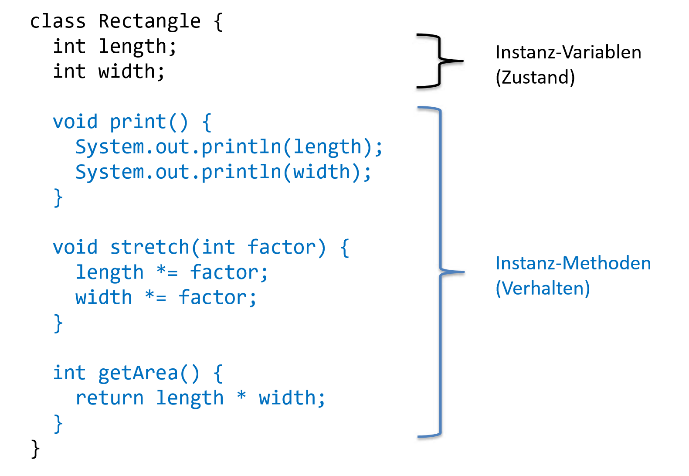
* Nur zu **schmaleren** Datentypen
  + Deshalb bei float immer das «f» angeben, ansonsten wird es als Double interpretiert und Double to float geht nicht!
  + Achtung kann zu Informationsverlust führen
    - Rundungsfehler 🡪 Float / double
    - Gleitkomma- zu Ganzzahl

**double** **d1** = 1.0/3.0;

**float** **f1** = (**float**) d1;

# Klassen und Objekte

Klasse

Eine Klasse ist der Bauplan, oder die Schablone, für Objekte.  
Eine Klasse beinhaltet verschiedene Zustände(Varibeln):

* private double length;
* private String name;
* etc.

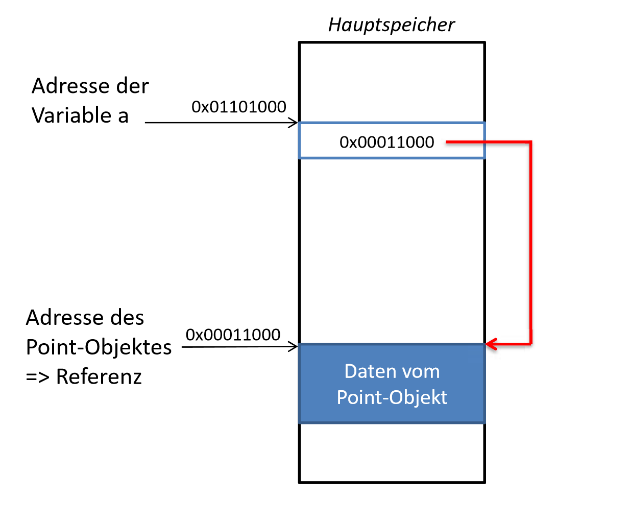
Und ebenfalls beinhaltet eine Klasse verschiedene Verhalten(Methoden):

* public double getLength()
* public changeName();
* etc.

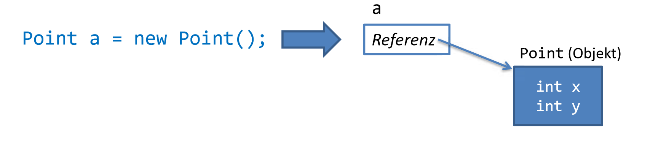
**Doppelleben einer Klasse**

Eine Klasse führt ein Doppelleben, sie kann fogendes sein:

* Datentyp
  + Point a
  + Car car1
* Schablone für Objekte
  + New Point();
  + New car(int anzRaeder)

**Referenz**

**Ausdrücke speichern nur eine Referenz auf ein Objekt.**

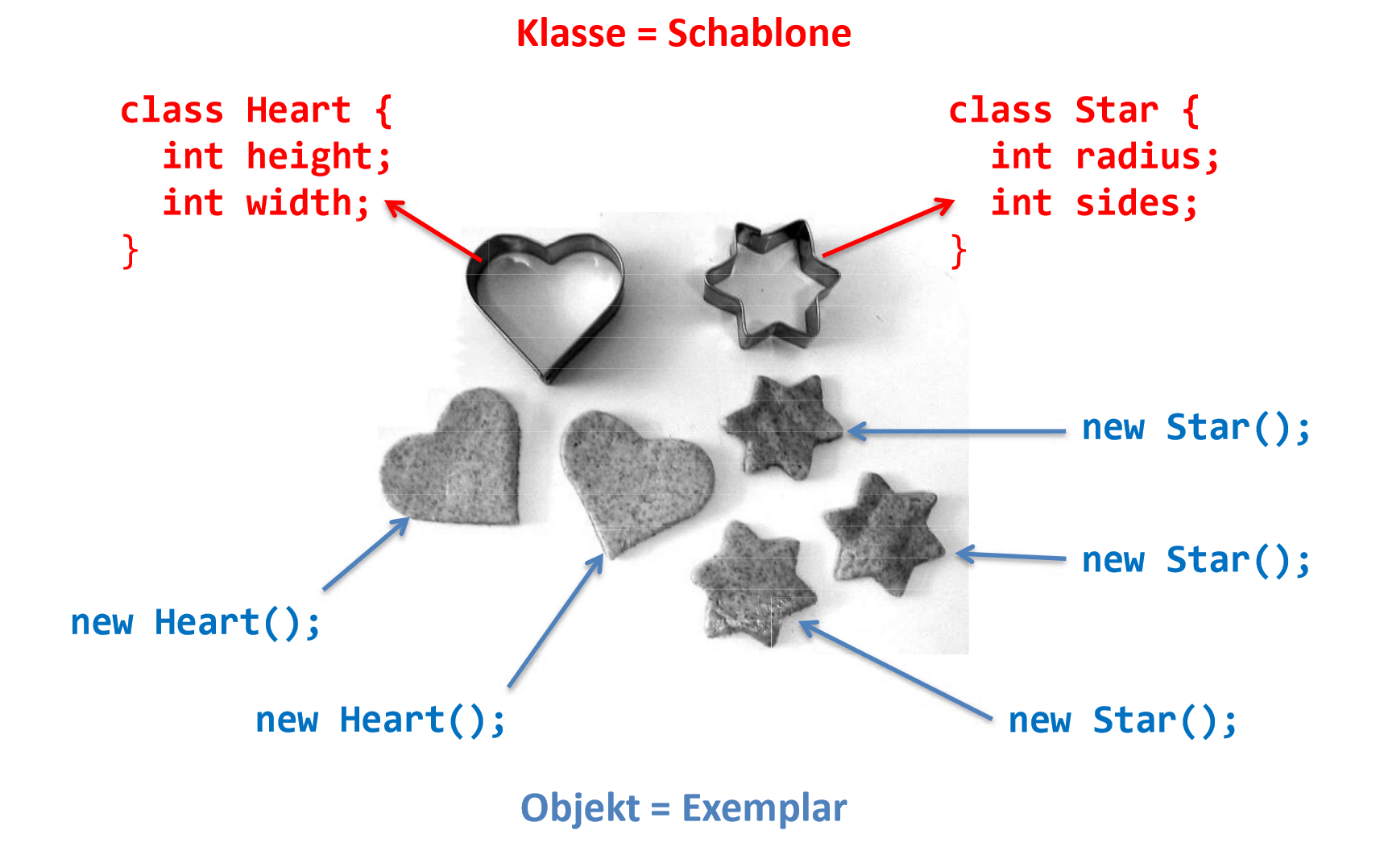


* **Null-Referenzen beachten**
  + **Wirft Null-Pointer Exception**
* **Point b = a;**
  + **Nur Referenzen werden kopiert**
  + **Gleiches Objekt!**

Objekte

Objekte werden aus Klassen erzeugt. Ein Objekt ist ein Exemplar(eine Instanz) einer Klasse.  
Bei der Instanzierung wird ein Objekt aus einer Klasse erzeugt, es wird also irgendetwas nach einem Bauplan entworfen.

* Point a = new Point();
* Point b = new Point();
* Objektname
* Klassenname(Bauplan)
* Variabel(Referenz)

Es können mehrere Instanziierungen stattfinden. Mehrere Objekte aus einer Klasse.

Methoden Aufrufe / this / Shadowing

**Car** **myCar** = **new** Car();

myCar.drive(100); //Geschwindigkeit wird als Parameter übergeben

**public** **class** **Car**

{

**private** **int** speed;

**public** **void** **drive**(**int** speed)

{

**this**.speed = speed; // Die Instanzvariabel wird = 100 gesetzt.

}

}

Lokale Variabel/Parameter verdeckt Instanzvariable, wenn Name gleich ist. Dies nennt man auch Shadowing.

Call by Value

Noch einmal kurz zurück zum Call by Value, da es wichtig ist. Aufpassen bei Methoden mit Referenz-Datentypen als Parameter(Arrays, Objekte,etc.):

1. Methode mit Parametern wird aufgerufen
2. Kopie vom Wert des Referenz-Datentyps wird als Argument an Methode übergeben
   1. Wert vom Referenz-Datentyp ist eine Adresse
   2. Daher wird die Kopie einer Adresse übergeben
   3. Änderungen des Objekts sind sichtbar
      1. Änderungen von Instanz Variabel möglich
      2. point1.setx(1); //Funktioniert
   4. Änderungen der Referenz sind unsichtbar(denn Referenz nur eine Kopie)
      1. Point1 = null // Funktioniert nicht den Referenz ist nur kopiert

**System**.***out***.println(rectangle1.getTopLeft().getX()); // Ausgabe = 8;

*changeTopLeft*(rectangle1);

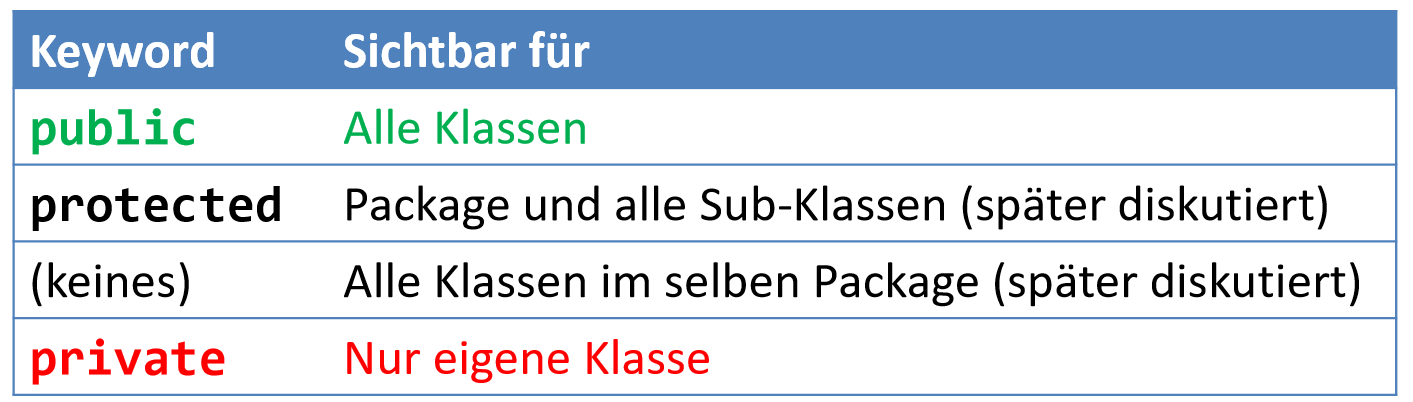
**System**.***out***.println(rectangle1.getTopLeft().getX()); // Ausgabe = 16

**private** **static** **void** **changeTopLeft**(**Rectangle** rectangle1) {

**Point** **links** = rectangle1.getTopLeft();

links.setX(links.getX() \* 2); //Änderungen am Objekt funktionieren

rectangle1 = **null**;//Änderungen an der Referenz keinen Einfluss(Kopie)

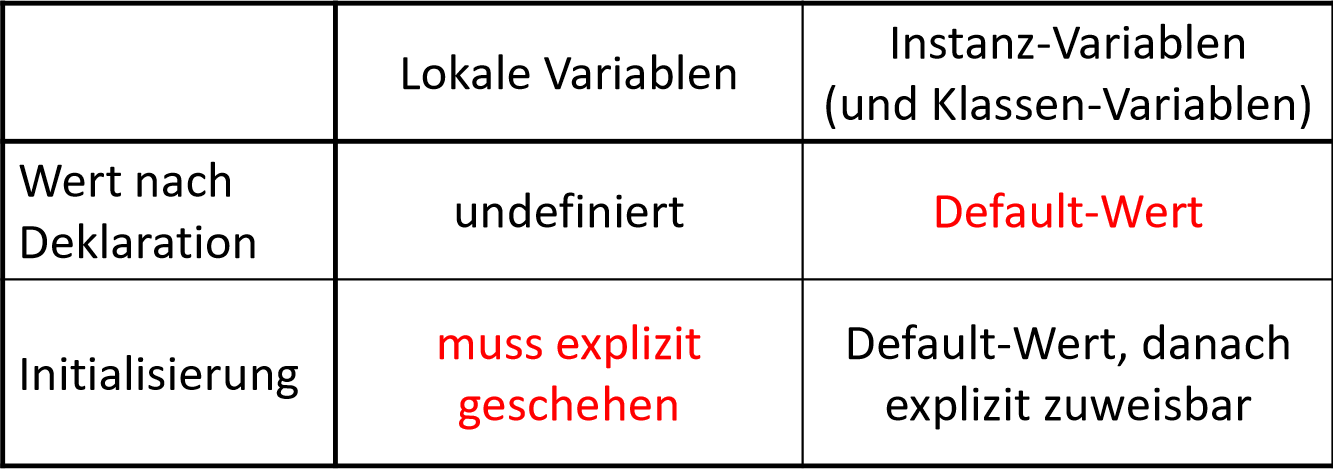
Zugriffkontrollen

**Verwendung(i.d.R)**

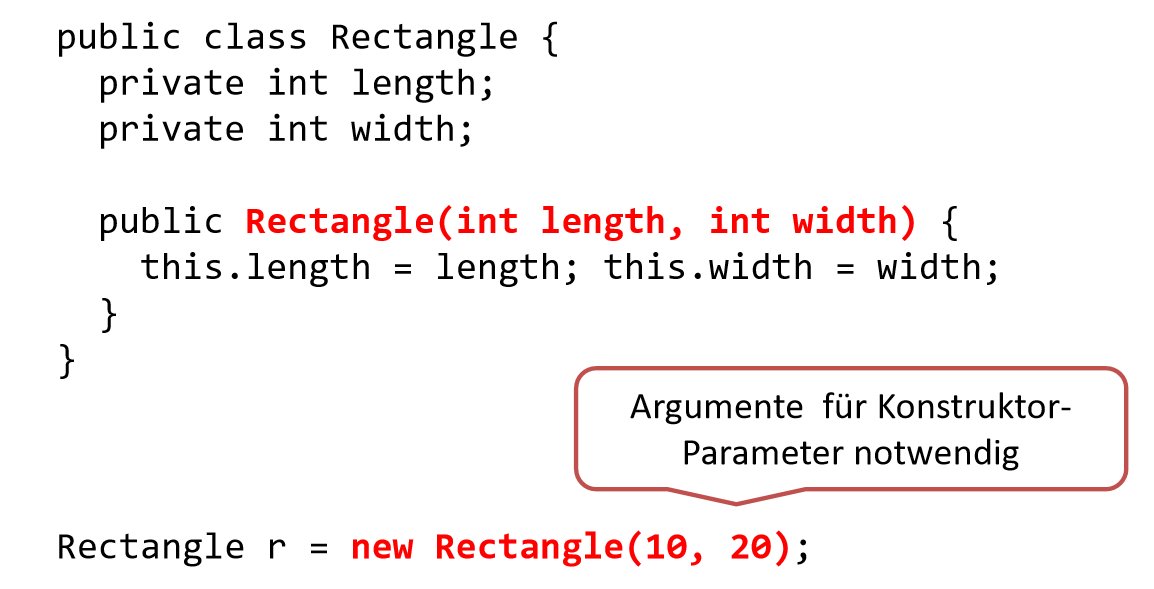
* **Instanzvariabeln private**
  + **Werden von Setter und Getter unterstützt**
* **Methoden public**

**Vorteil:**

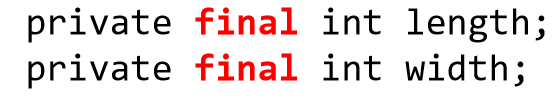
* + **Instanz Variabel Read-Only**
  + **Werte bei Setter prüfen**
  + **Werte umrechnen**
  + **Statistiken**

Initialisierung

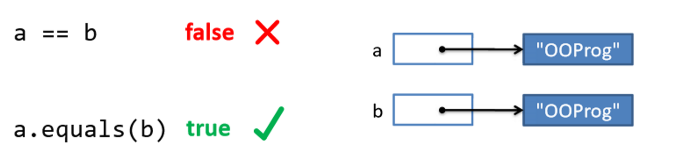
Konstruktur

Spezielle Methode zum Initialisieren von Objekten. Muss bei der Erstellung eines Objektes mitgegeben werden:

Final

Die Bedingung «final» bedingt, dass Werte von Instanzvariablen einmal gesetzt und danach nicht wieder verändert werden können.

# Spezielle Referenztypen

String

* Vordefinierte Klasse(Anstatt char[])
* String vergleichen mit equals-Methode
  + Gleiche Strings können auf andere Objekte verweisen
  + Mit == Referenzvergleich 🡪 kann false ergeben(z.T richtig wegen Compiler-Optimierungen)
  + .equals vergleicht Inhalt

**Vergleichen**

**String** **a** = "Hallo";

**String** **b** = "Hallo";

**System**.***out***.println(a == b); // kann false ergeben

**System**.***out***.println(a.equals(b)); // true

**Kopieren**

**String** **a** = "Hallo";

**String** **b** = **new** String(a);

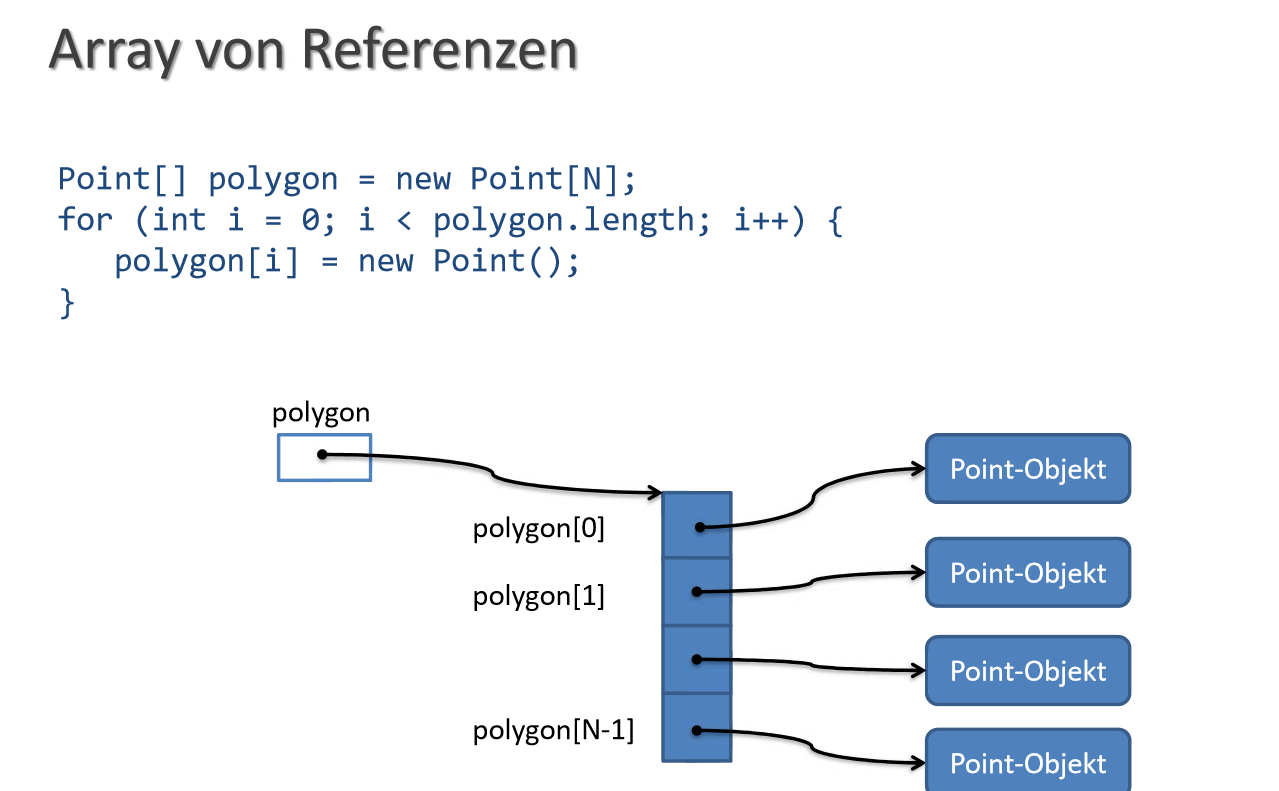
Oder  
b = a;

**Zusammenhängen**

a += b

«b» kann dabei ein String sein oder eine Char c.

Arrays

**Arrays von Referenzen/Objekte**

**Vergleichen**

Bei Array will man wieder den Inhalt vergleichen und nicht die Referenzen, deshalb braucht man wieder die equals-Methode:

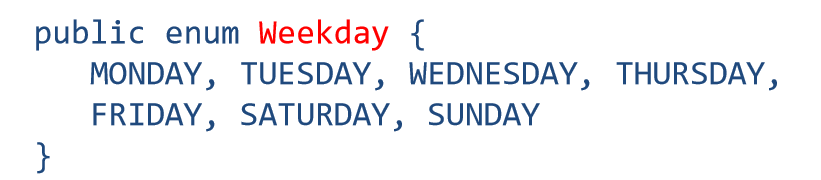
**char**[] **charArray1** = {'e','i','n','s'};

**char**[] **charArray2** = {'e','i','n','s'};

**System**.***out***.println(**Arrays**.*equals*(charArray1, charArray2)); //true

Enums

Eigenere Datentype mit Auflistung von Werten, ähnlich wie eine Tabelle:



**private** **enum** *Level* {

***LEVEL1***, ***LEVEL2***, ***LEVEL3***

}

*Level* currentLevel = *Level*.***LEVEL2***;

Es sind nur die fixen Werte «LEVEL1, LEVEL2, LEVEL3» auswählbar. ENUMs sind eher selten.

**Switch mit ENUM**

**switch** (currentLevel)

{

**case** ***LEVEL1***:

**System**.***out***.println("Beginner");

**break**;

**case** ***LEVEL2***:

**System**.***out***.println("Mid");

**break**;

**case** ***LEVEL3***:

**System**.***out***.println("Expert");

**break**;

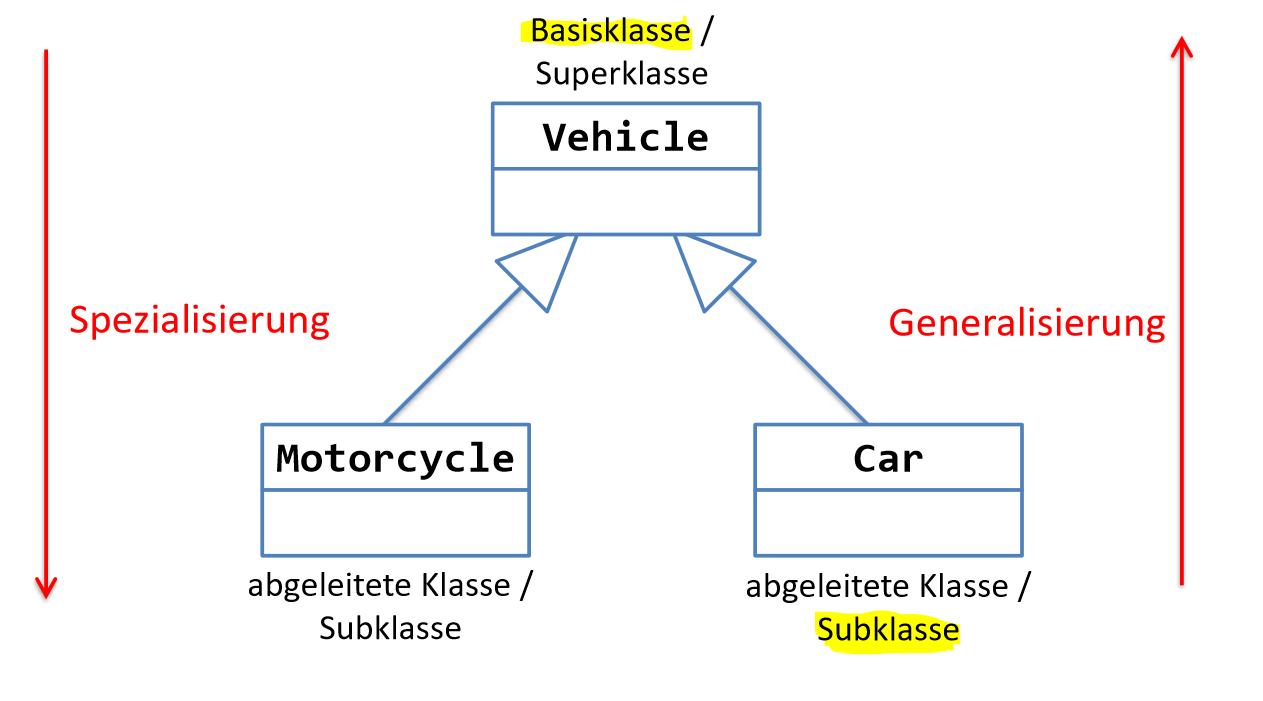
}

**IF mit ENUM**

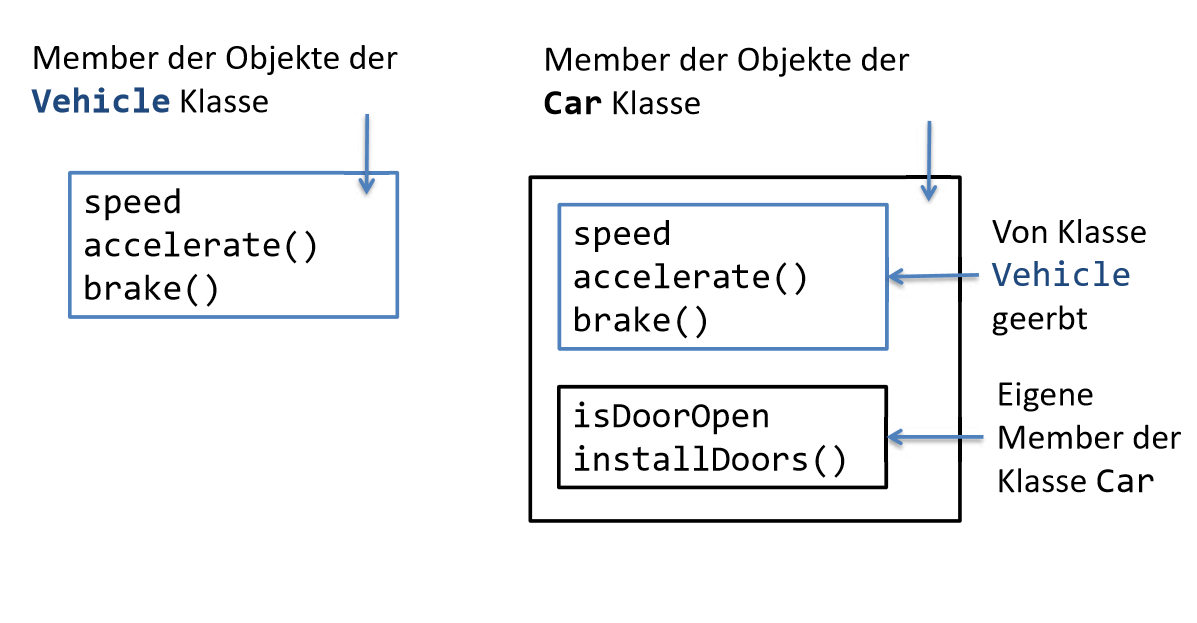
**if**(currentLevel == *Level*.***LEVEL1***){

**System**.***out***.println("Beginner");}

# **Vererbung**

Beziehung zwischen zwei Klassen für logische Spezialisierung

**Unterklasse**

Eine Unterklasse erbt alle Zustände und Verhalten der Basisklasse. Das bedeutet die Subklasse erhält alle Methoden und Instanzvariablen der Superklasse:

**Anwendung**

|  |  |
| --- | --- |
| **public** **class** **Vehicle**  {  **private** **int** speed;    **private** **void** **accelerate**(**int** speed)  {  **this**.speed = speed;  }  } | **public** **class** **Car** **extends** **Vehicle**  {  **private** **boolean**[] isDoorOpen = **new** **Boolean**[5];  **public** **void** **openDoors**()  {  **Arrays**.*fill*(isDoorOpen, **false**);  }  } |

**public** **class** **VehicleTest**

{

**public** **static** **void** **main**(**String**[] args)

{

**Car** **car** = **new** Car();

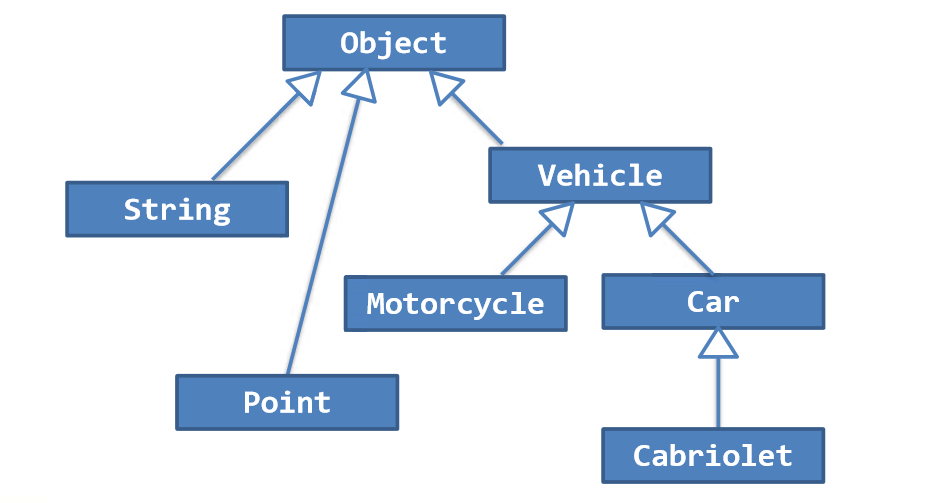
car.accelerate(100);

**System**.***err***.println(car.getSpeed());

car.openDoors();

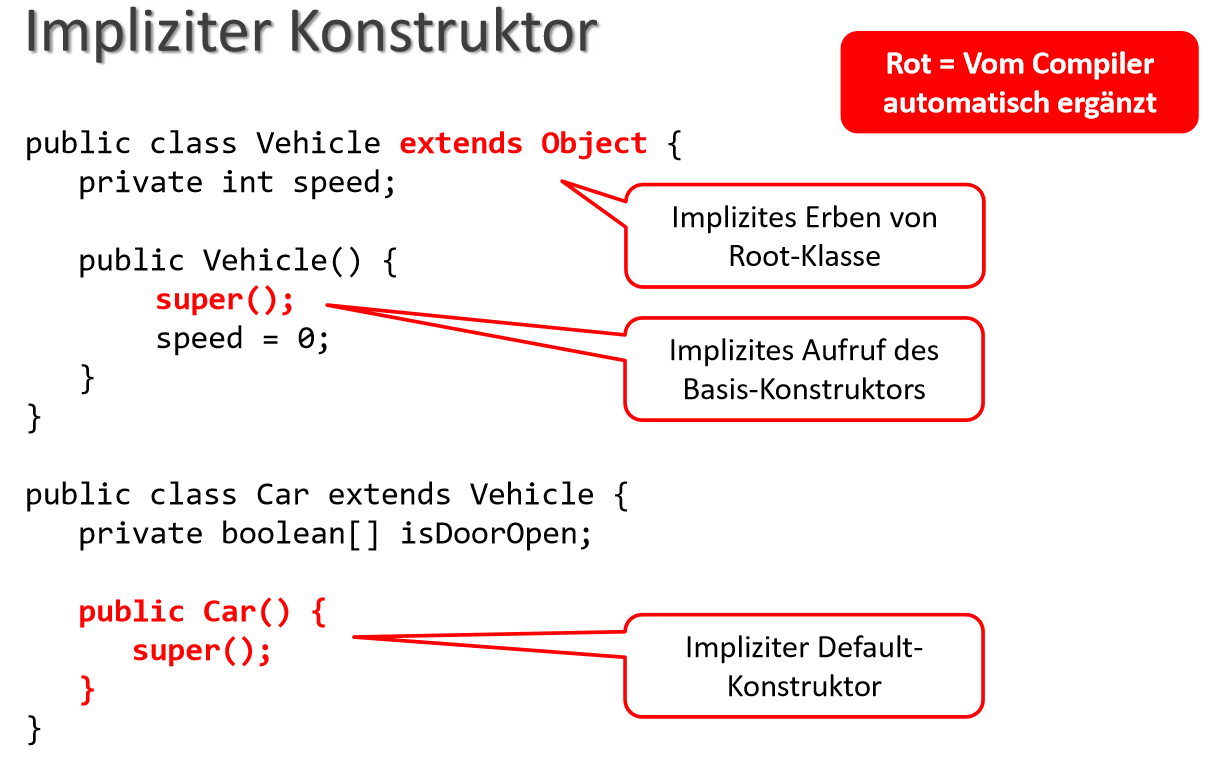
}

}

**Struktur**

**Merke: Object ist immer die oberste Basisklasse. Es erben also alle die Members von Object.**

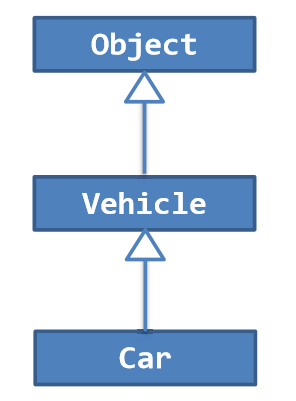
Impliziter Konstruktor

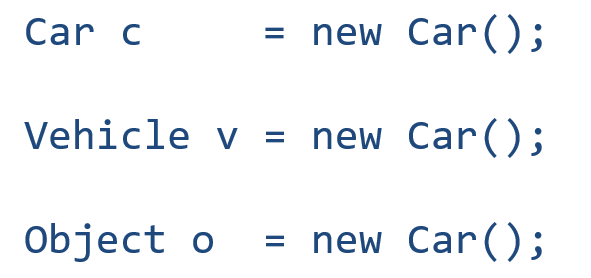
Eine Subklasse erbt automatisch den Konstruktor der Superklasse.

Super() ruft den Konstruktor der Basisklasse auf und muss als erstes Statement stehen. Braucht der Basis-Konstruktor irgendwelche Parameter, müssen diese auch beim super(Parameter) der Unterklasse angegeben werden:

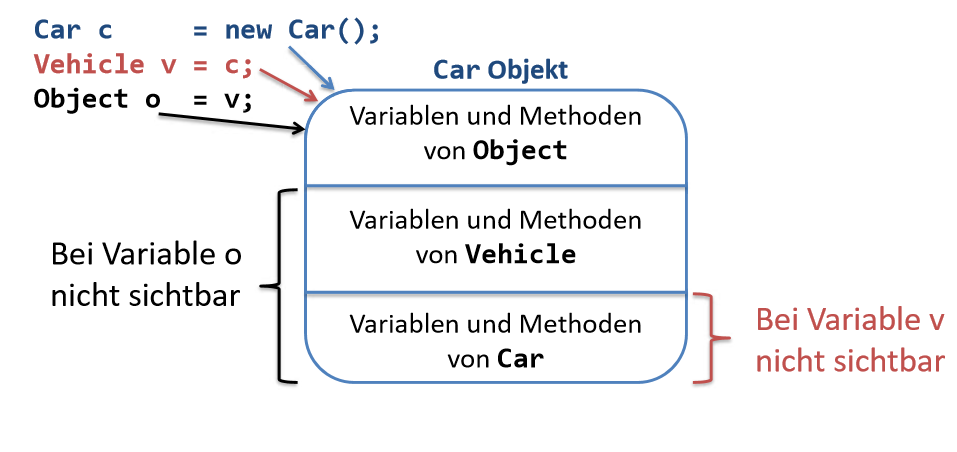
|  |  |
| --- | --- |
| **public** **Vehicle**(**int** speed)  {  **this**.speed = speed;  } | **public** **Car**(**int** speed)  {  **super**(speed);  **Arrays**.*fill*(isDoorOpen, **false**);  } |

# Polymorphismus

Ein Objekt hat nicht nur den Typ seiner Klasse, sondern auch die Typen seiner Superklassen.



**Sichten**

* **Jedes Objekt kann verschiedene Sichten haben.**
* **Sicht wird von (Daten)Typ gegeben**
* **Je nach Sicht können Variablen/Methoden von einer Klasse gesehen werden oder nicht**
  + **Es handelt sich um ein Car-Objekt vom Typ Vehicle.**
  + **Kann Variablen und Methoden von Vehicle und Object sehen.**
  + **Sieht die Variablen vom Car nicht.**

**Vehicle** **v** = **new** Car(0);

v.accelerate(100);

v.openDoor() // nicht möglich sieht Methode nicht

Object o = **new** Car(0);

o.accelerate(10) //nicht möglich, sieht Methode nicht

# **Overriding & Dynamic Dispatch**

**Überschreiben**

**Wie oben erläutert wurde, werden alle Methoden der Superklasse an die Subklasse vererbt. Nun kann es aber auch sinnvoll sein, dass es eine Methode in der Subklasse gibt, welche den gleichen Namen trägt, jedoch eine andere Funktion hat.  
Man kann also eine geerbte Methode überschreiben, dies nennt man Overriding:**

|  |  |
| --- | --- |
| **public** **class** **Vehicle**  {  **public** **void** **report**()  {  **System**.***out***.println("Vehicle");  }  } | **public** **class** **Car** **extends** **Vehicle**  {  ***@Override***  **public** **void** **report**(){  **System**.***out***.println("Car");  }  **}** |

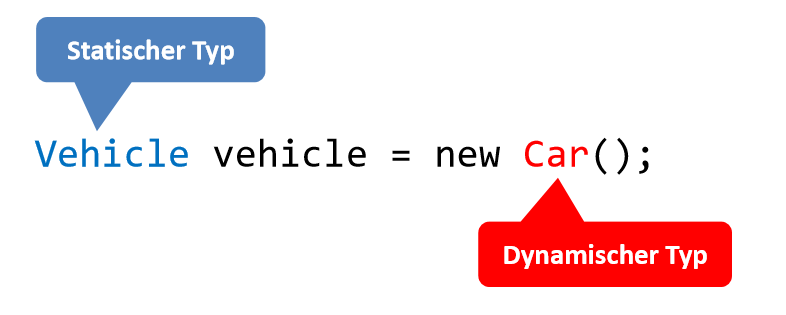
* **@Overriding**
  + **Stellt sicher, dass es Methode in Basisklasse**
  + **Wenn weggelassen, dennoch überschrieben**
  + **Empfohlen anzugeben**

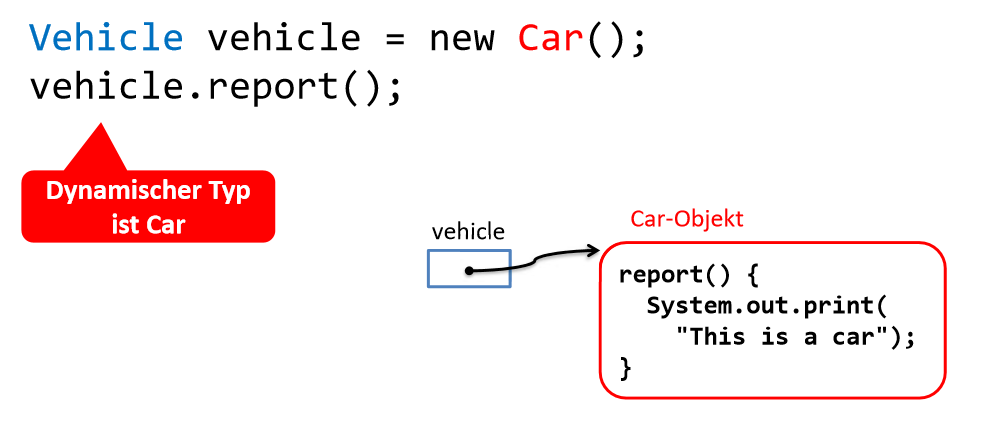
**Methoden der Basisklasse aufrufen**

**Mit super.<Methodenname> kann eine Methode der Basisklasse aufgerufen werden. Dies macht in vielen Overriding Methoden Sinn. Denn meistens will man zur Basismethode noch zusätzliche Logik hinzufügen:**

|  |  |
| --- | --- |
| **public** **class** **Vehicle**  {  **public** **void** **accelerate**(**int** speed)  {  **this**.speed = speed;  }} | ***@Override***  **public** **void** **accelerate**(**int** speed)  {  closeDoors(); **super**.accelerate(speed);  } |
|  |  |

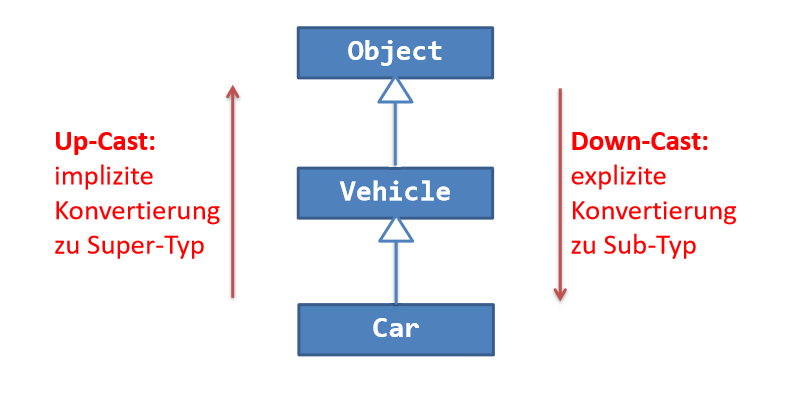
**Bevor man beim Auto abfahren kann müssen noch die Türen geschlossen werden.**

**Dynamic Dispatch / Dynamische Bindung**

**Beim Aufrufen von nicht-privaten Instanzmethoden wird die Methode gemäss dem dynamischen Typ(Typ während Laufzeit) aufgerufen:**

**Dynamischer Typ gemäss Deklaration zu Compile-Zeit.**

**Up- & Down-Casts**



|  |  |
| --- | --- |
| **Implizite Up-Casts** | **Explizite Down-Casts** |
| Car c = new Car(0);  Vehicle v = c;  Object o = v; | Object o = new Car(0);  Vehicle v = (Vehicle) o;  Car c = (Car) v; |

**Aufpassen bei Null-Referenzen**

**Dynamischer-Typ prüfen**

Mit «instanceof» kann man prüfen, ob eine statische Variabel = dem dynamischer Variabel entspricht.

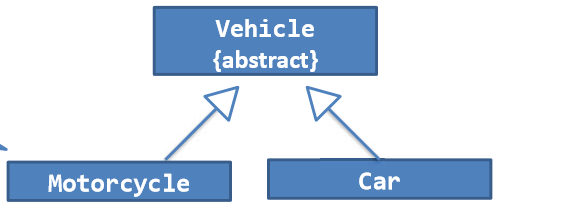
**Vehicle** **v** = **new** Car(0);

**Object** **o** = **new** Vehicle(0);

**System**.***err***.println(v **instanceof** **Car**);

**System**.***err***.println(o **instanceof** **Car**);

# **Abstrakt**

Abstrakte Klassen

Eine abstrakte Klasse ist eine Klasse, welche nicht instanziert werden kann. Somit erzwingt man, dass man eine Instanziierung von Unterklassen machen muss.

|  |  |
| --- | --- |
| **Abstrakte Klasse** | **Test Klasse** |
| **public** **abstract** **class** **Vehicle**  {  **private** **int** speed;  **public** **Vehicle**(**int** speed)  {  **this**.speed = speed;  } | **Vehicle** **v** = **new** Car(0);// Möglich  **Object** **o** = **new** **Vehicle**(0);  **//Instanzierung von Vehicle nicht möglich da diese Klasse abstrakt ist!** |

Alle Subklassen von Vehicle erben die Variabeln und Methoden.

**Abstrakte / Finale Methoden**

* **Kein Rumpf, nur Deklarationen**
* **Methoden in Subklasse implementiert werden**

|  |  |
| --- | --- |
| **Abstrakte Methoden** | **Subklassen** |
| **public** **abstract** **class** **Vehicle**  {**public** **abstract** **void** **report**();} | ***@Override***  **public** **void** **report**(){  **System**.***out***.println("Car");} |

**Final Methoden**

**Es gibt auch finale Klassen und Methoden:**

|  |  |
| --- | --- |
| **Abstrakte Methoden** | **Subklassen** |
| **public** **final** **void** **stop**()  {  speed = 0;  }  **report**(); | **public** **final** **class** **Vehicle**  **{**  **}** |

**Eine finale Methode kann nicht überschrieben werden.**

**Eine finale Klasse kann keine Subklassen haben.**

**Quiz**

Computergenerierter Alternativtext:
Quiz 
Letzte Woche
OO HS2017 -Schnittstellen
4
class Base {
public void print() {
printKind();
printDetail();
}
public void printKind() { 
System.out.println("TYPE Base"); 
}
private void printDetail() { 
System.out.println("DETAIL Base"); 
}
}
class Derived extends Base {
public void printKind() { 
System.out.println("TYPE Derived"); 
}
private void printDetail() { 
System.out.println("DETAIL Derived"); 
}
}
Base o = new Derived();
o.print();
Erwartete Ausgabe?


Lösung Quiz: Es wird «DETAIL BASE» ausgegeben, weil die Methoden privat sind!

# Bildschirmausschnitt**Schnittstellen**

Grundlagen

Problem bei Vererbung von Klassen: Subklasse kann nur von einer Basisklasse abgeleitet sein. Ansonsten würde es zu Problemen führen:

* Namenskonflikte
  + Gleiche Memberdeklaration in Basisklassen
* Diamant Problem
  + Mehrfach geerbte Instanzvariabeln(int speed)
* Compiler-Komplexität

Die Lösung sind Schnittstellen:

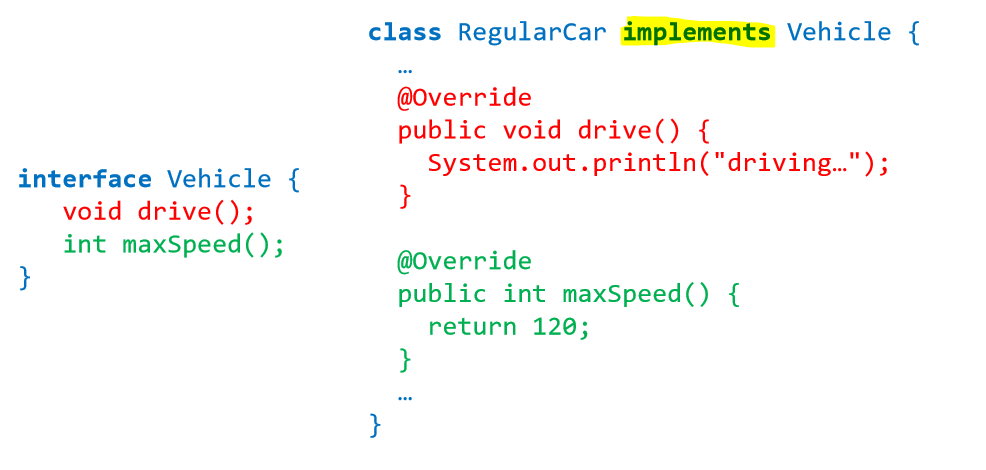
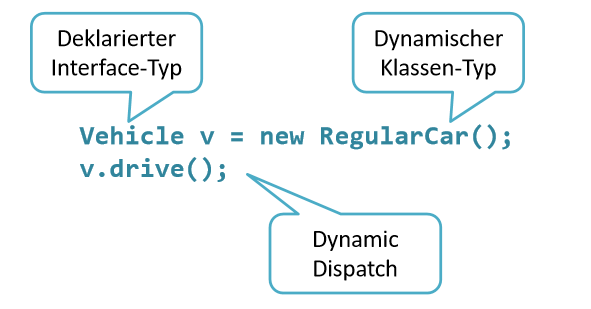
Schnittstellen ermöglichen Mehrfachvererbung

Interface beschreibt nutzbare Funktionalität einer Klasse:

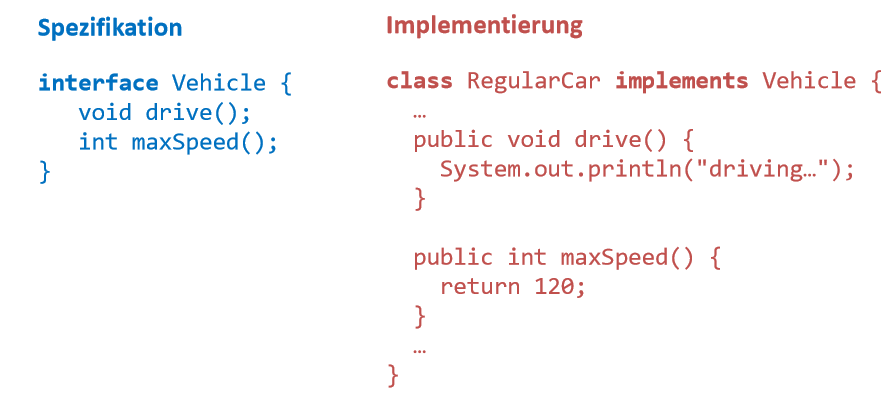
* Wohnwagen
  + Vehicle🡪Fahren
  + House 🡪Wohnen

Grundlagen

Vertrag zwischen Klasse und seiner Aussenwelt:

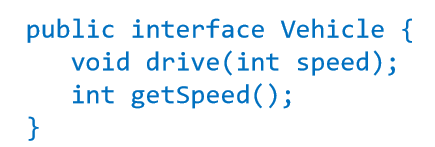
* Klasse muss diese Funktionalität implementieren
* Aussenwelt darf diese Funktionalität nutzen
* Jede Schnittstelle kann auch ein Typ sein
* Schnittstellen nicht instanzierbar

Konzept

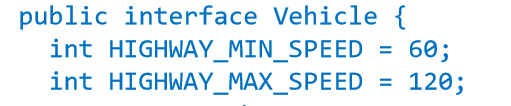
Das Konzept hinter Interfaces ist, die Spezifikation und die Implementierung getrennt ist.

Lose Kopplung

* Benutzer verwendet nur Schnittstelle
  + Implementierung austauschbar
  + Erweiterbarkeit
* Unabhängige Entwicklung
  + Team 1 Schnittstellen x-y
  + Team 2 Schnittstelle z

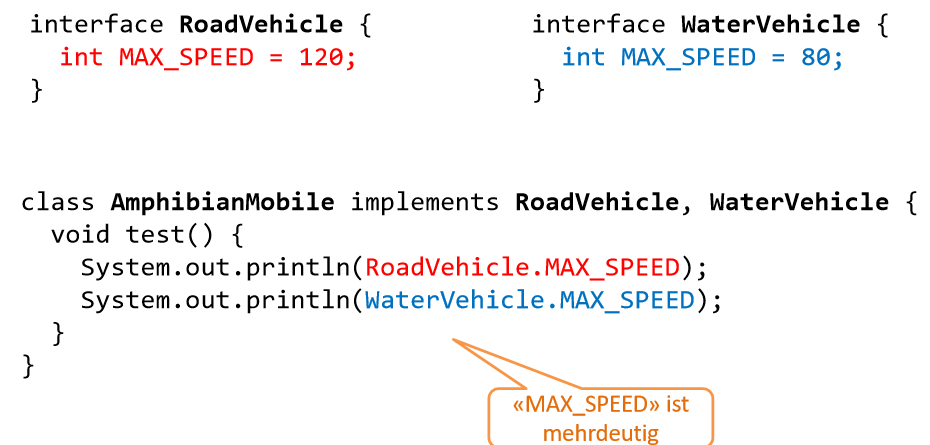
Syntax

* Methoden sind implizit public und abstract
* Interface implizit abstract

Konstanten

* Grossbuchstaben
* Automatisch public static final

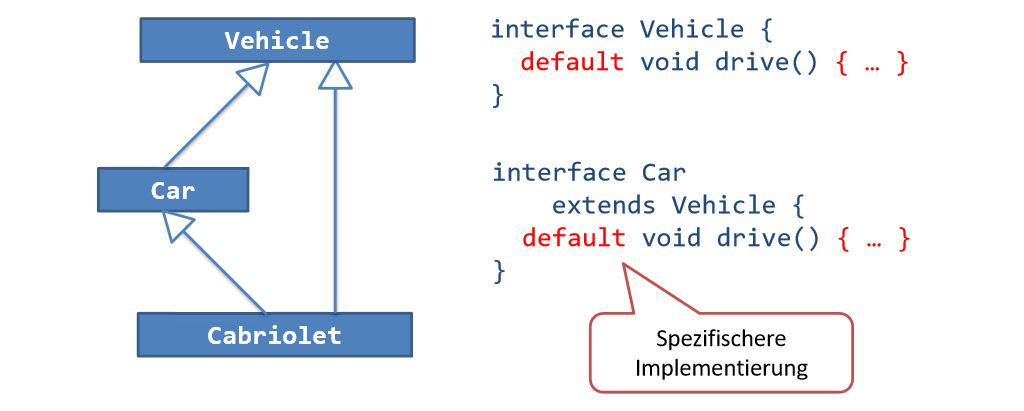
Mehrfach Implementationen

* Gleiche Methoden
  + Methode wird nur einmal implementiert
* Gleiche Konstanten
  + Konstanten werden mit <Interfacename>.<Konstantenname> eindeutig identifiziert

Default Methods

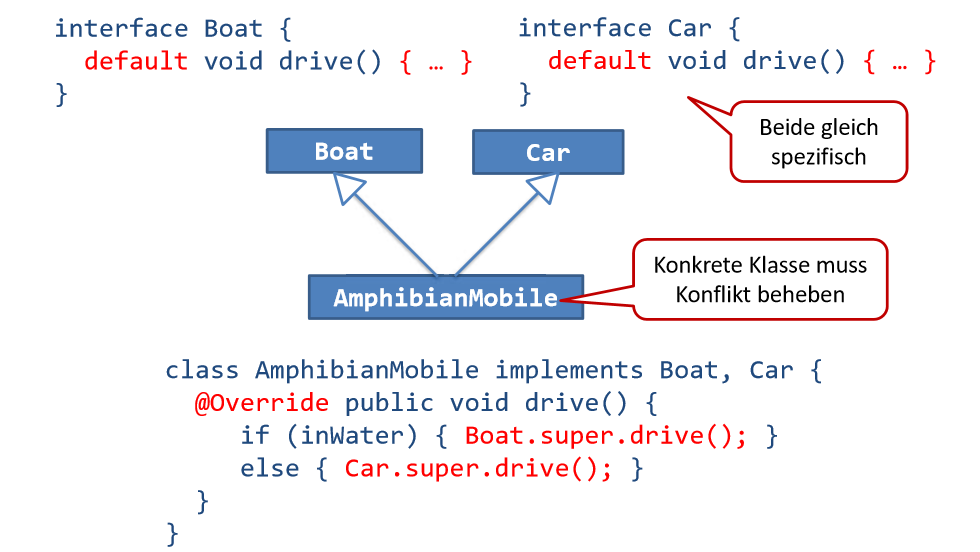
Bei Interfaces können auch «default Methoden» eingebunden werden. Diese können, müssen aber nicht, überschrieben werden von der Klasse.

**Mehrere Default Methoden**

Gib es mehrere Sub Interfaces mit mehreren Deault-Methoden, dann wird die nächst höher gelegene Default-Methode verwendet:

Hier würde also die drive-Methode von Car verwendet werden . Denn immer die nächst «gelegene» Default-Methode ist die spezifischere Methode.

**Konflikt mit Default Methoden**



Abstrakte Klassen vs. Interfaces

|  |  |
| --- | --- |
| Gleich(=) | Ungleich(=) |
| Nicht instanzierbar | Interfaces mehrfach Implementation |
| Abstrakte Methoden | Interfaces keine Variablen |
| Konstanten | Klassen Basis Konstruktor |

# Statische Deklarationen

Static Variable / Klassenvariabelm

Eine statische Variable operiert nur auf der Klasse und ist kein Teil der Instanz. Deshalb wird diese aufgerufen mit:

Classname.variable

Es ist daher wie eine globale Variabel.

**static** **int** *numberDoor*; //Deklaration statische Variabel

**Car**.*numberDoor* = 4; //Aufruf Variable

Static Methode /Klassenmethoden

Eine statische Methode operiert nur auf der Klasse und ist somit auch kein Teil der Instanz. Gleicher Aufruf wie bei der statischen Methode:

**static** **void** **numberDoor**()

{

**System**.***out***.println("A cas has number of doors: " +*numberDoors*);

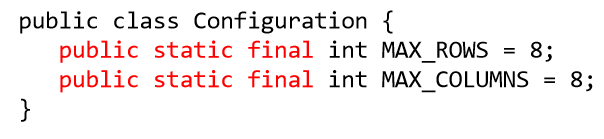
};

**Car**.*numberDoor*();

Überblick:

Konstanten in Objekten

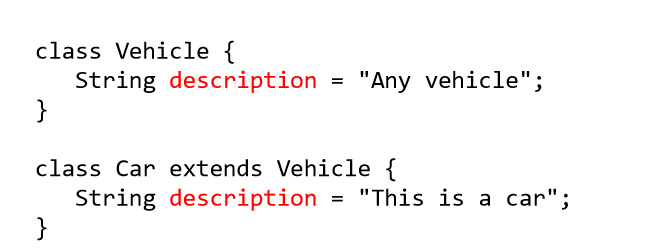
Konstanten in Objekten sind Variabeln, welche public, static und final sind, ebenfalls alles grossgeschrieben:



Statische Bindung

**Variabel Hiding**

**Variabel Hiding nennt man den Vorgang, wenn in der Subklasse Instanzvariable mit gleichem Namen erstellt, welche es in der Basisklasse bereits gibt.**



**Statische Bindung**

**Mit der Statischen Bindung ist es möglich, dass Variabel Hiding «aufzulösen»:**

**Die Variabel der Basisklasse darf nicht private sein!**

# Exception Handling

Exception werden verwendet um Ausnahmen oder Fehler anzugeben.

* Logische Programmierfehler
  + NullPointerException
  + Stack Overflow
  + / by Zero
  + Arrax out of Bounce
* Fehler in Bedingungen und Eingabe
  + NextInt() 🡪 man darf keine Buchstaben eingeben
* Probleme mit der JVM
  + Kein Speicher mehr, etc.

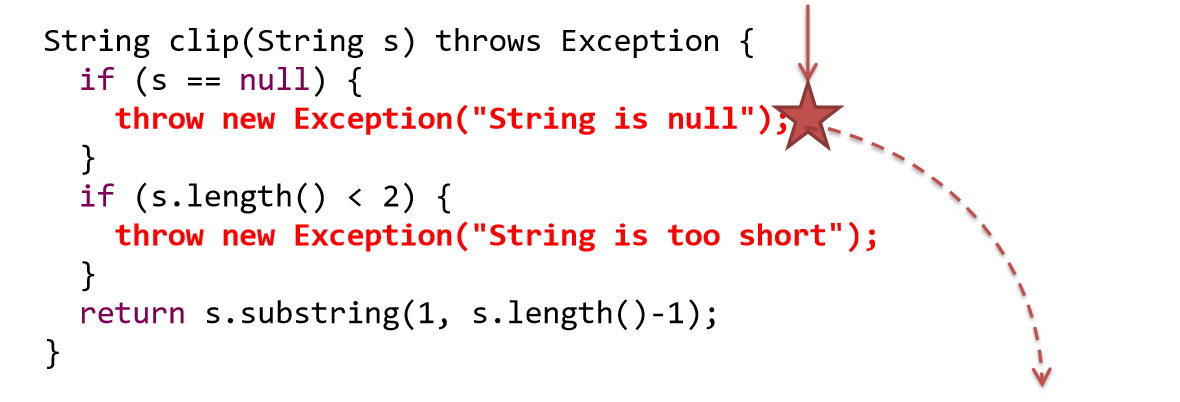
**Auslösen von Exceptions**

|  |  |
| --- | --- |
| **Caller Methode** | **Callee Klasse** |
| **public** **static** **void** **main**(**String**[] args) **throws** **Exception**  {  **Car** **c** = **new** Car();  c.setCarName(""); | **public** **class** **Car** **extends** **Vehicle**  {  **private** **String** carName;  **public** **void** **setCarName**(**String** carName) **throws** **Exception**  {  **if** (carName == **null**)  {  **throw** **new** Exception("Is NUll"); }  **else**  {  **this**.carName = carName; }  } |

* Beide Methoden müssen Signatur(Methodenkopf) anpassen
  + Angegeben werden, welche Exception(s) es geben kann
  + Trhows ExceptionType1, Exception2
* Exception werfen
  + Es muss angegeben werden, welche Exception(s) geworfen wird

**Throw-Statement**

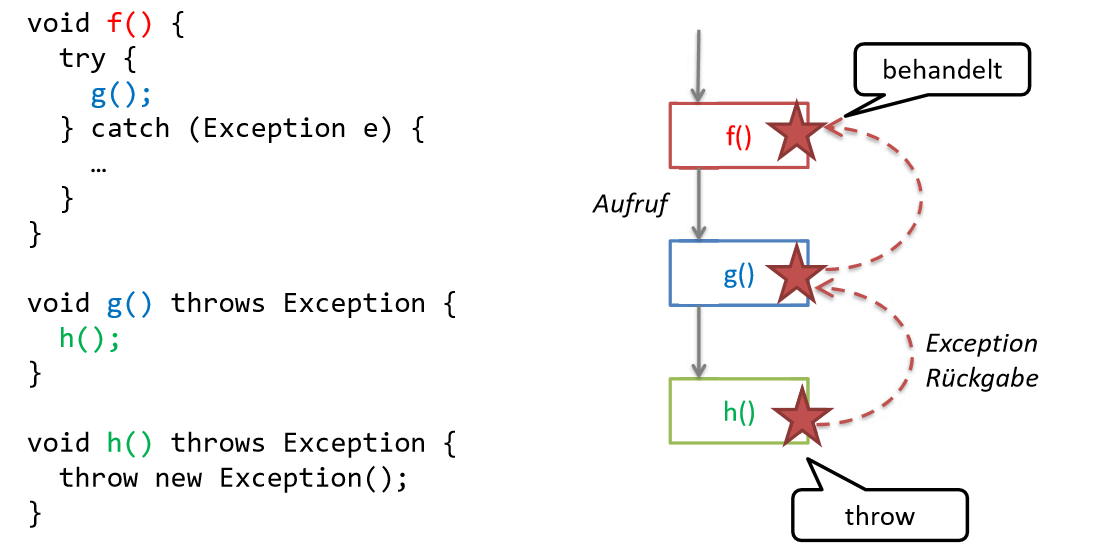
* **Unterbricht aktuellen Code**
* **Verlässt die Methode(wenn nicht behandelt)**
* **Exception beim Caller ausgelöst**
  + **Leitet Exception zur Aufrufer-Methode weiter**



**Exceptions behandeln / try - catch**

**Mit dem try-Catch Block können Exception behandelt werden:**

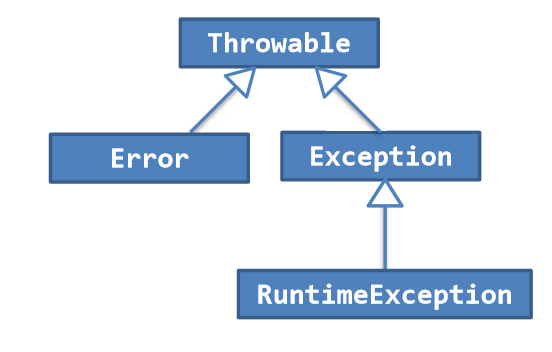
|  |  |
| --- | --- |
| **Caller Methode(welche Exception behandelt)** | **Callee Klasse** |
| **try**{  **Car** **c** = **new** Car();  **String** **s** = **null**;  c.setCarName(s);  } **catch** (**NullPointerException** **e**)  {  **System**.***out***.println("String NULL");  }  **catch** (**Exception** **e**)  {  **System**.***err***.println("Other exception");  } | **public** **class** **Car** **extends** **Vehicle**  {  **private** **String** carName;  **public** **void** **setCarName**(**String** carName) **throws** **Exception**  {  **if** (carName == **null**)  {  **throw** **new** NullPointerException("Is NUll"); }  **else**  {  **this**.carName = carName; }  } |



**Solange Exception nicht behandelt werden, sende Exception weiter zum Aufrufer!**

**Wenn kein try, catch vorhanden ist, wir sich die JVM um den Error kümmern.**

**Unterschied Error – Exception**

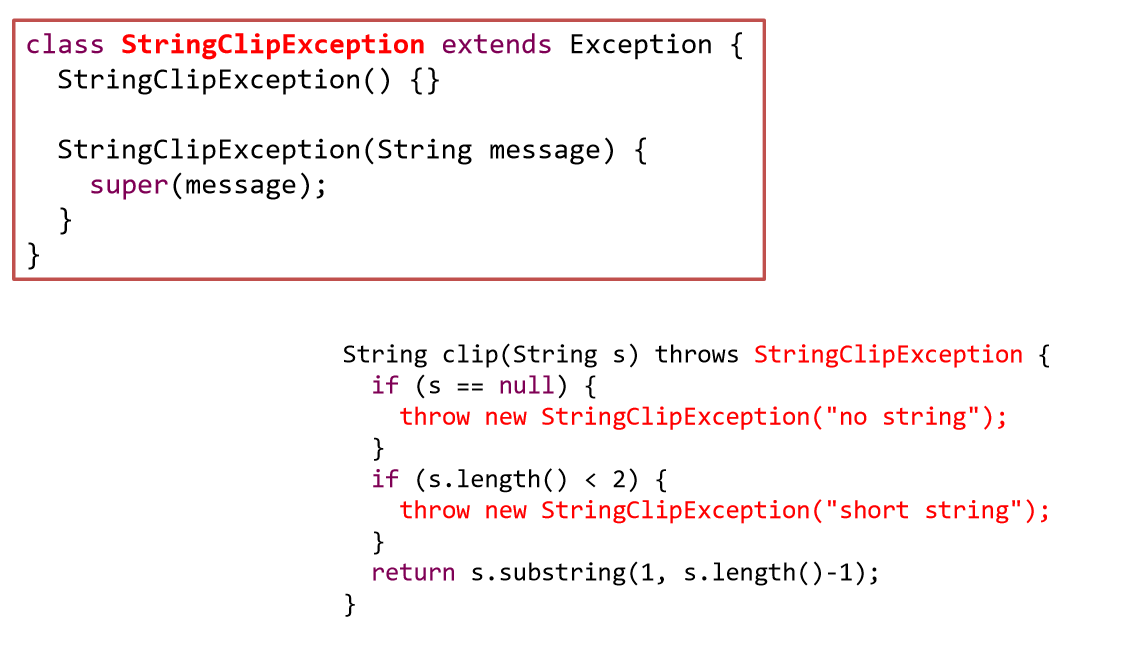
* Error
  + Schwerwiegender Fehler
  + Normale Anwendung soll/kann Fehler nicht behandeln
* Exception
  + Laufzeitfehler, welcher Anwendung behandeln kann/soll
  + Fehler in Eingabe, Parameter, etc.

Vordefinierte Exception

Es gibt eine Menge vordefinierte Exception, die wichtigsten für uns sind:

* IllegalArgumentException – Ungültiger Wert für Parameter
* NullPointerException – Dereferenzieren von null
* [IndexOutOfBoundsException](https://docs.oracle.com/javase/7/docs/api/java/lang/IndexOutOfBoundsException.html#IndexOutOfBoundsException()) – Etwas ausserhalb vom Bereich(bsp Array)

Benutzerdefinierte Exception

Ebenfalls ist es möglich selber Exception zu definiere, hierbei sollte man es aber nicht übertreiben:

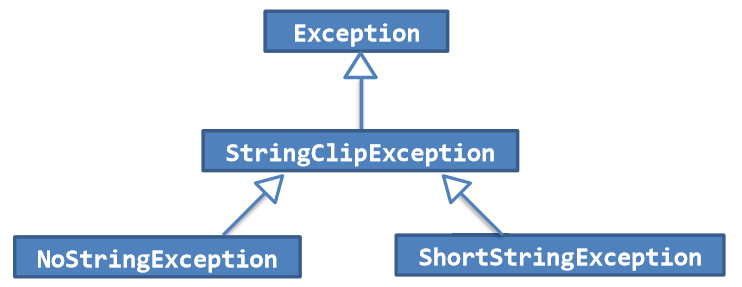
Checked & Unchecked Exceptions

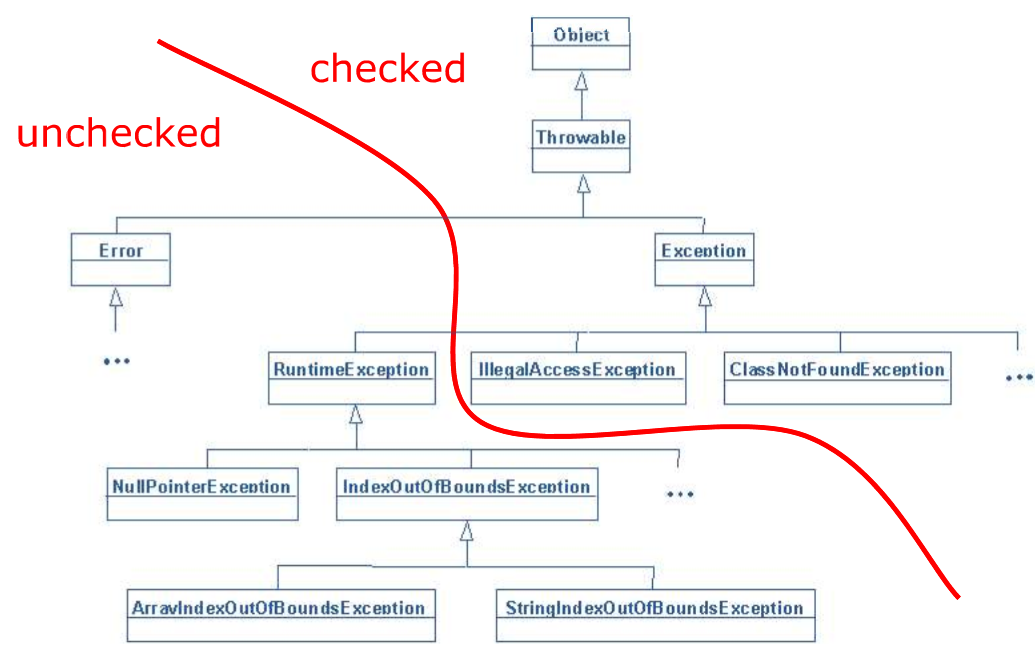
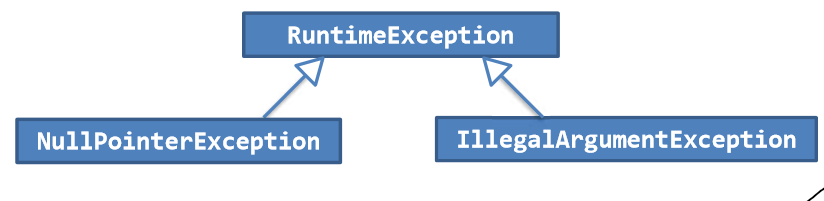
**Checked Exceptions**

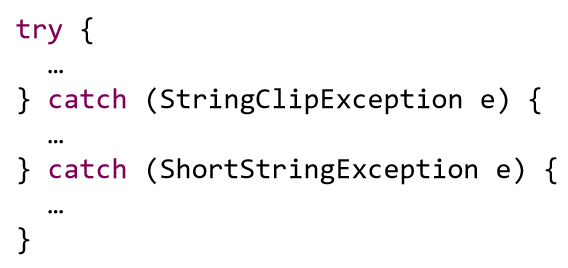
**Checked Exceptions sind Exceptions, welche behandelt oder weitergeworfen werden müssen. Sie brauchen also immer ein:**

* **Throws <Exception> im Methoden Kopf + throw Anweisung**
* **try-catch**

**Unchecked Exceptions**

**Unchecked Exception sind Exceptions, welche nicht behandelt werden müssen. Sie brauchen daher auch keine throws-Deklaration oder Behandlung**



**Mehrere Catch-Klauseln**

**Wie oben erwähnt, ist es möglich, mehrere Catch-Klauseln einzubauen. Hier muss man aber aufpassen, dass es keinen toten Code gibt. Toter Code entsteht dann, wenn eine der oberen Catch-Klauseln eine Exception fängt, welche eine Basisklasse ist:**

**Syntax**

|  |  |
| --- | --- |
| **Variante 1** | **Variante 2** |
| **try**{  **…**.  }**catch** (**NullPointerException** **e**)  {  **System**.***out***.println("String NULL");  }  **catch** (**StringIndexOutOfBoundsException e**)  {  **System**.***err***.println("Other exception");  } | **try**{  **…**.  } **catch** (**NullPointerException** | **StringIndexOutOfBoundsException** **e**)  {  **System**.***out***.println("Exception: " +e);  **}** |

**Auch bei der zweiten Exception muss aufgepasst werden, dass kein toter Code entsteht.**

**Geschachtelte try-Blöcke**

Ebenfalls ist es möglich, mehrere try-Blöcke einzubauen. Wenn der erste Catch die Exception nicht auffängt, kann diese bei 2. Catch-Block aufgefagen werden:

**try**

{

…

**try**{

…

} **catch** (**StringIndexOutOfBoundsException** **e**)

{

**System**.***out***.println("Innere Exception: " + e);

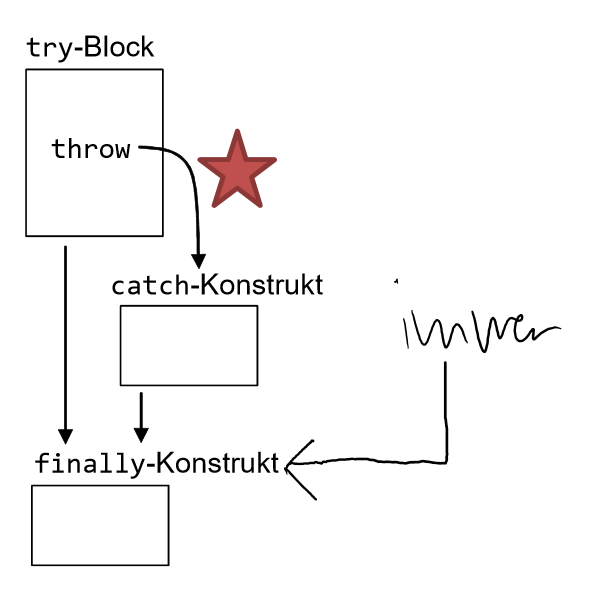
}

} **catch** (**NullPointerException** **e**)

{

**System**.***out***.println("Äusserer Exception: " + e);

}

**Finally-Block**

**Der finally-Block steht jeweils am Ende und wird immer ausgeführt.**

**try**

…

} **catch** (**Exception** **e**)

{

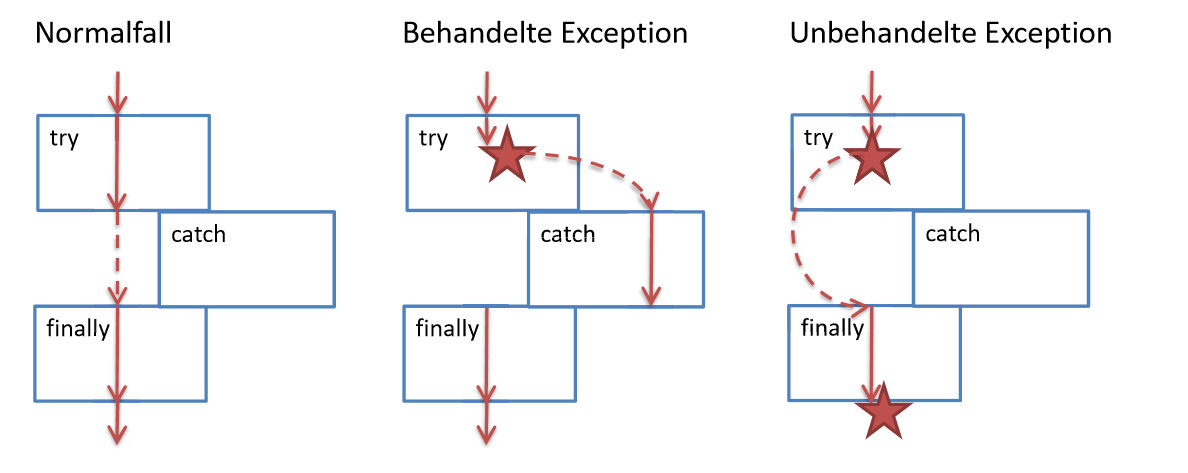
**System**.***out***.println(" Exception: " + e);

} **finally**

{

**System**.***err***.println("Finish");

}



**Auch kann man das finally ohne catch nutzen(besser aber try-with Ressources):**

**try-with Ressources**

Wenn man Objekte hat, welche geschlossen werden müssen, rät es sich, diese in ein try-with-Ressources einzupacken. Somit werden diese automatisch am Schluss des try-Blocks geschlossen:

**try**(**Scanner** **scanner** = **new** Scanner(**System**.***in***))

{

**System**.***out***.println("Print String");

**String** **s** = scanner.nextLine();

}

Wird empfohlen bei Scanner, oder SQL-Anweisungen, etc.

# Package

Befehle

* Paketnamen kleingeschrieben
* Umgekehrter Domänen na,e

**Package ch.hsr.lectures;**

**import ch.hsr.computers;**

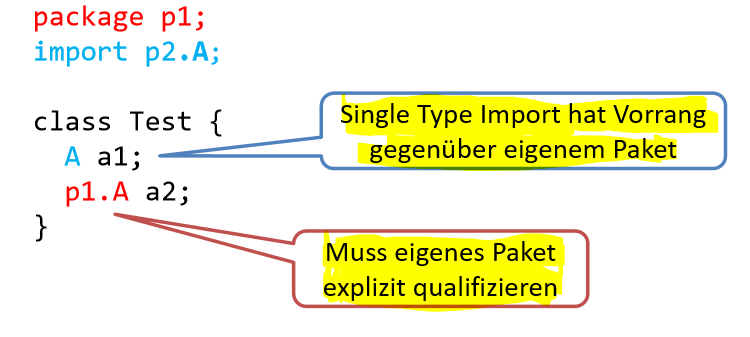
**import ch.htw.\***

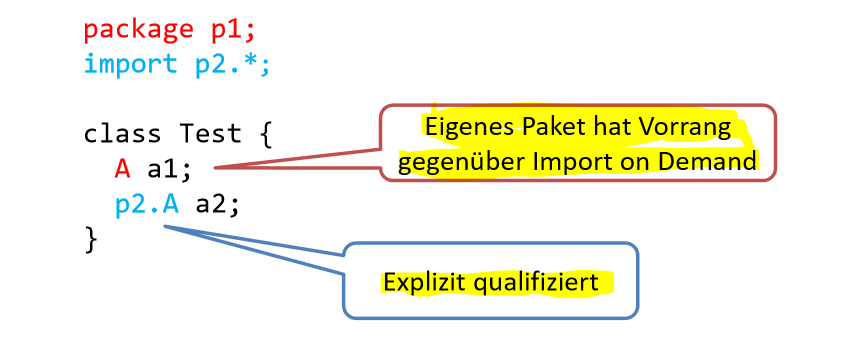
FQN Fully Qualified Name

Der FQN einer Klasse lautet:

* <packagename>.<Subpackagename>.<Classname>

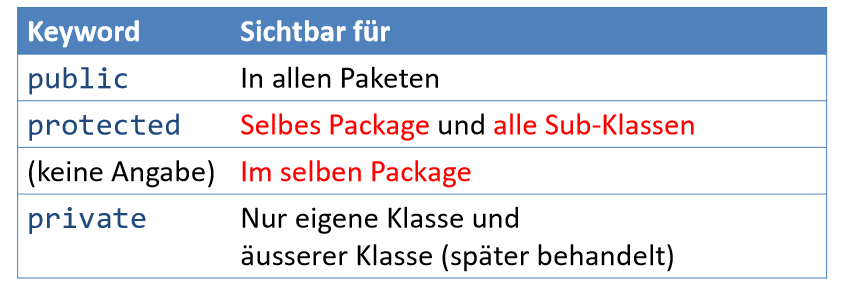
Namenskonflikte

Merke: Eigenes Paket hat Vorrang gegenüber Import, ausser es ist ein Single-Import:



Haben zwei Importe die gleiche Variabel gibt es ein Fehler. Dann muss man den FQN benutzen:

<Packagename>.<Variabel>

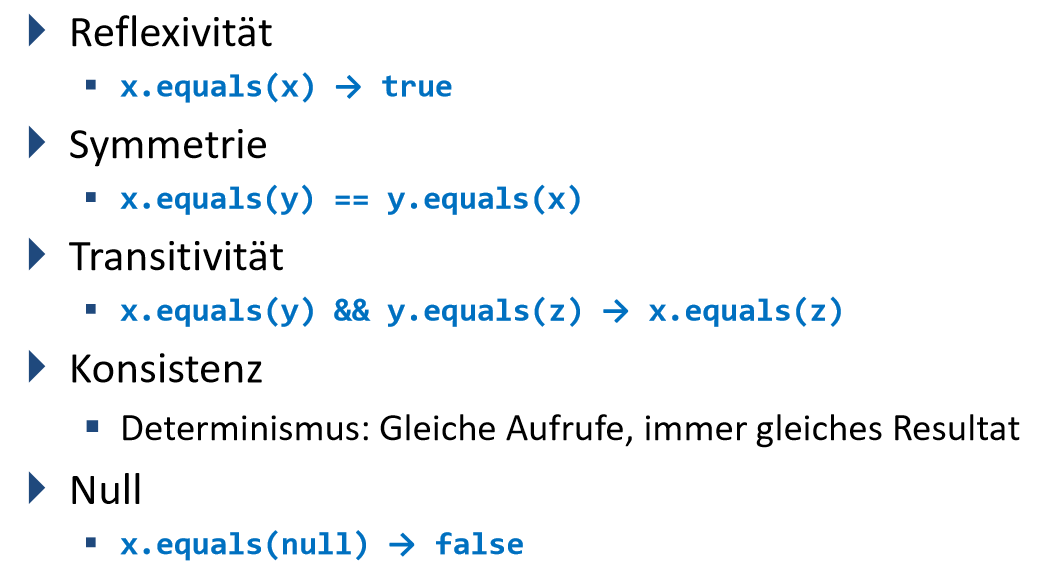
Sichtbarkeit

# Objekt Grundfunktionen

**Gleichheit / Vergleichen**

Objekte müssen mit der equals-Methode verglichen werden. Jedoch muss man diese für Objektvergleiche überschreiben. Nur für String kann man die Standard-Equals-Methode verwenden:

|  |  |
| --- | --- |
|  |  |
| ***@Override***  **public** **boolean** **equals**(**Object** obj)  {  **if** (getClass() != obj.getClass() || obj == **null**)  {  **return** **false**;  }  **else**  {  **Person** **other** = (**Person**) obj;  **return** (**this**.firstName.equals(other.getFirstName()) && **this**.lastName.equals(other.getLastName()));  }  } | ***@Override***  **public** **boolean** **equals**(**Object** obj)  {  **if** (!(**super**.equals(obj)))  {  **return** **false**;  }  {  **Student** **other** = (**Student**) obj;  **return** **this**.studentNumber == other.studentNumber;  }  } |



Achtung! Hash Codes bei equals Methode beachten!



Wenn die Equals-Methode überschrieben wird, muss auch die HashCode-Methode überschrieben werden.

**Sortieren**

**Standard Arrays**

**Die Sortierung von Standard-Arrays(Char, Int) ist kein Problem:**

|  |  |
| --- | --- |
| **int**[] **intArray1** = {2,1,4,3};  **int**[] **intArray2** = {4,3,2,1};    **Arrays**.*sort*(intArray1);  **Arrays**.*sort*(intArray2) | **char**[] **cA1** = {'h','a','l','l','o'};  **char**[] **cA2** = {'w','e','l','p','e'};    **Arrays**.*sort*(cA1);  **Arrays**.*sort*(cA2 |

**Achtung: Beim Sortieren von char Arrays werden Grossbuchstaben höher priorisier als Kleinbuchstaben!**

**Arrays von Objekten**

**Arrays von Objekten lassen sich nicht mehr so einfach sortieren. Dafür braucht es einige Dinge:**

* **Klasse bedingt «implements Comparable<Klasse>»**
* **Compare Methode muss implementiert sein**

**compareTo-Methode**

* **<0: this ist kleiner als other**
* **>0: this ist grösser als other**
* **0: this ist gleich other**

**Aufpassen dass es keine «NullPointerException» gibt und keine Überläufe.**

|  |
| --- |
| **public** **class** **Person** **implements** Comparable<Person>  {  …  ***@Override***  **public** **int** **compareTo**(**Person** other)  {  **int** **result** = lastName.compareTo(other.LastName());  **if** (result == 0)  {  result = firstName.compareTo(other.FirstName());  }  **return** result;  }  } |
| **public** **class** **Student** **extends** **Person** **implements** Comparable<Person>  {  …  ***@Override***  **public** **int** **compareTo**(**Person** o)  {  **Student** **other** = (**Student**) o;  **return** **this**.studentNumber - other.studentNumber;  }  } |

**Sortieren**

**Sobald man die compareTo Methode implementiert hat, sind die Arrays auch sortieren:**

**Person**[] **personArray** =

{ (**new** Person("Severin", "Dellsperger")), **new** Person("Marco", "Rida"),**new** Person("ogna", "loop") };

**Arrays**.*sort*(personArray);

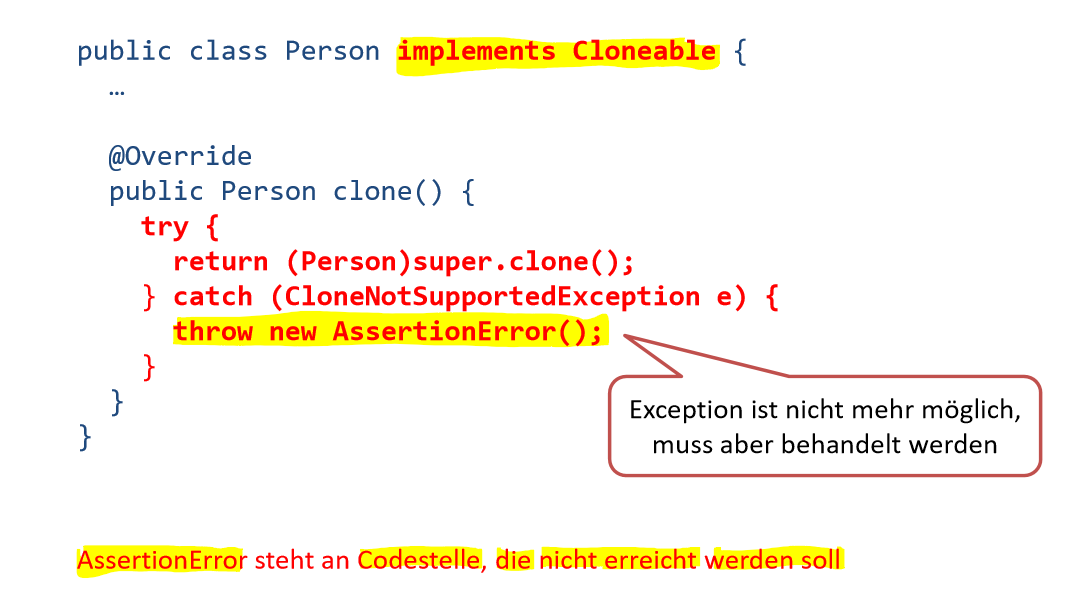
**for** (**Person** **person** : personArray)

{

**System**.***out***.println("" +person.getLastName() +" " +person.getFirstName());

}

**Klonen**

Mit der clone()-Methode lassen sich Objekte Kopien von Objekten erzeugen. Diese Methode muss jedoch wieder zuerst überschrieben werden:

# Design Prinzipien

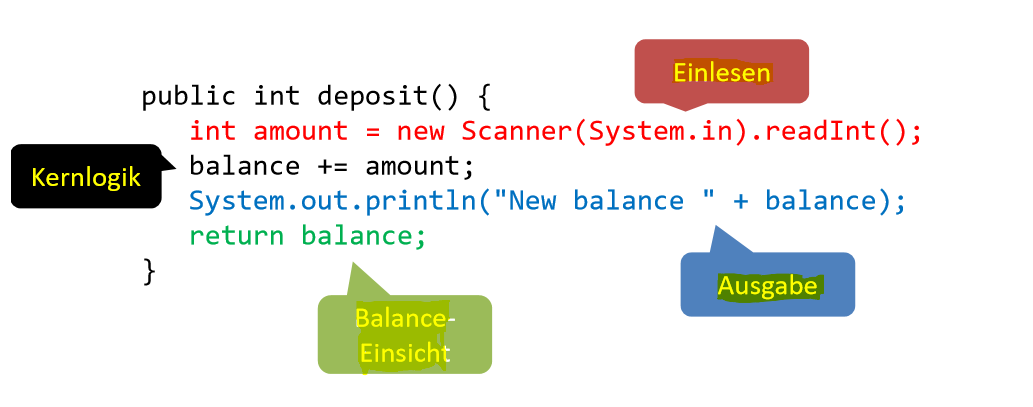
Design Prinzipien ist ebenfalls ein wichtiges Thema, denn es wichtig das man korrekten und schönen Code schreibt. Die zwei wichtigsten Themen lauten:

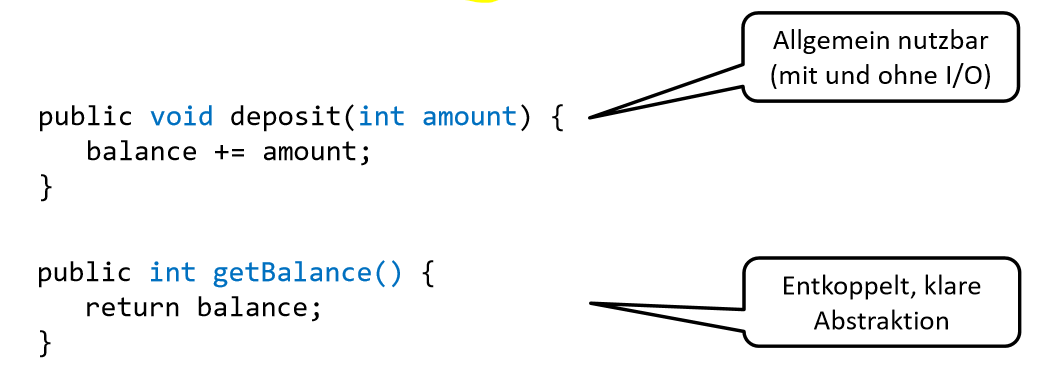
* Hohe Kohäsion
  + Logische zusammengehörige Aspekte bündeln(Eingabe, Ausgabe, etc.)
* Lose Kopplung

Diese Schritte passieren auf Stufe Methode, Klasse und Paket.

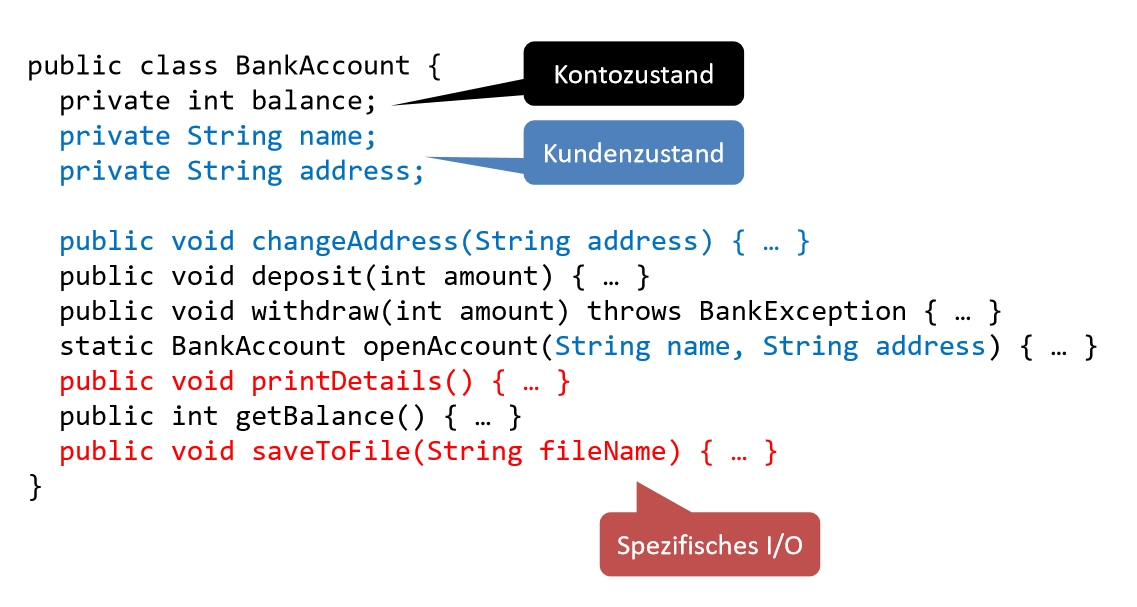
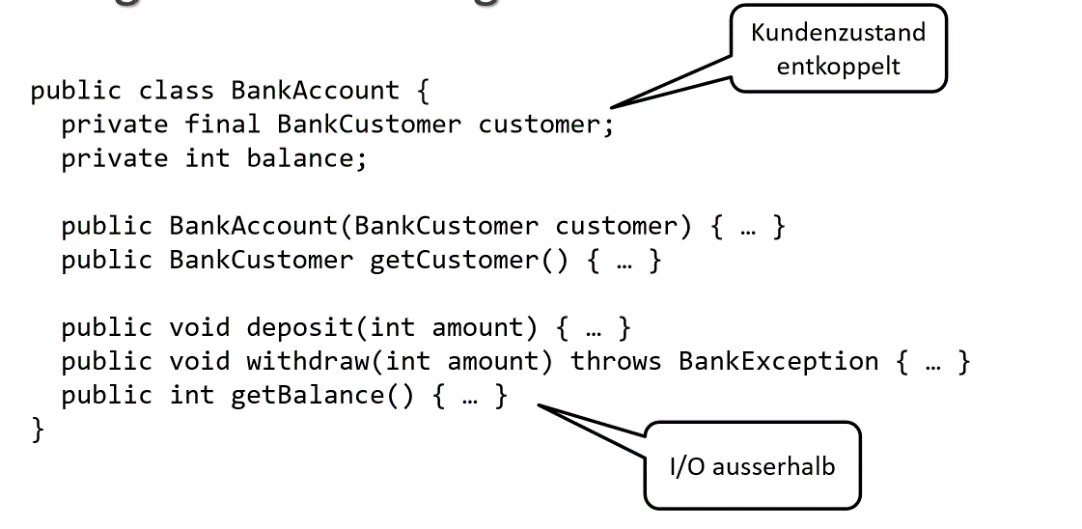
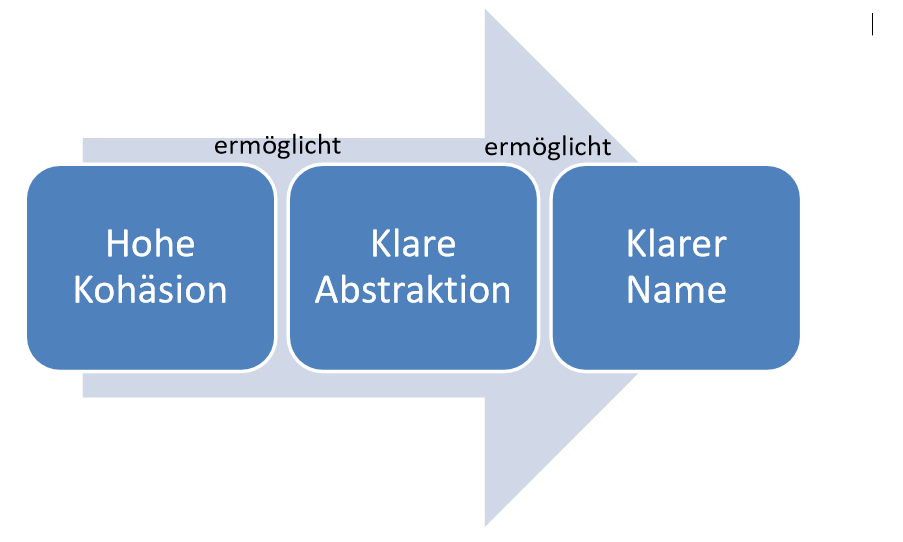
Kohäsion der Methode

Bei der Methode ist es so, dass man logische Teile trennen sollte:



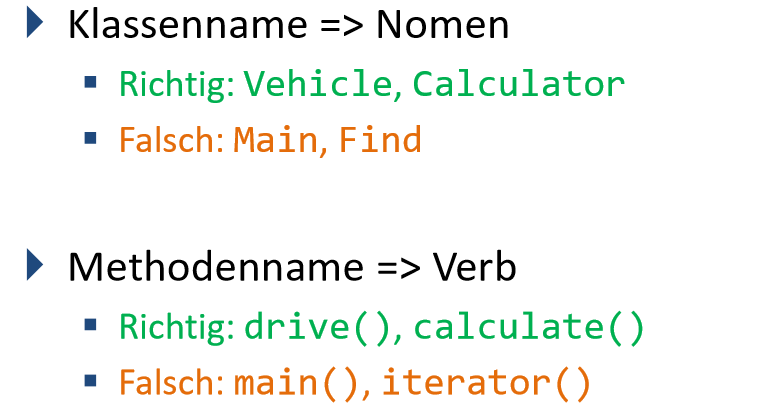
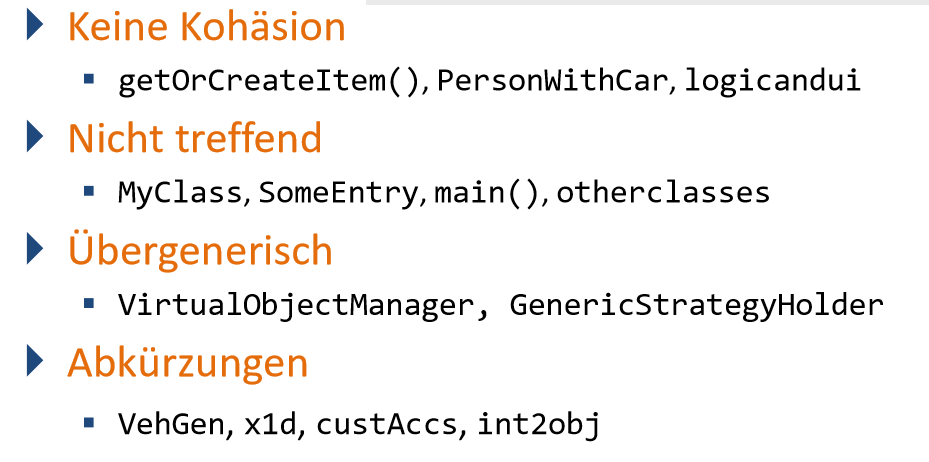


Kohäsion der Klasse

Auch bei der Klasse sollte man logische Teile auseinander halten:

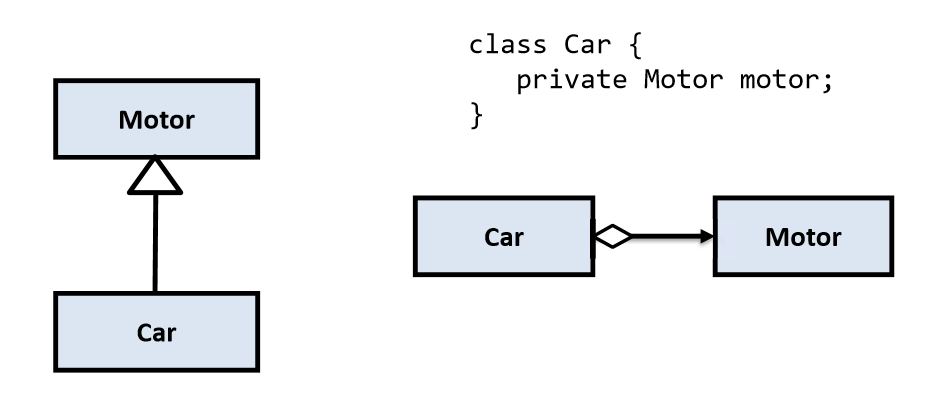
Klare Benennung

Es ist sinnvoll, alle Objekte klar zu benennen, auch wenn das nicht immer so einfach ist.



Vererbung / Aggregation

* Vererbung nur bei Typ-Polymorphismus
  + «is-a» Beziehung
  + Gleiches bei Schnittstellen
  + «extends / implements»
* Aggregation (Bevorzugt)
  + «has-a» Beziehung
  + Gewöhnliche Referenz auf ein anderes Objekt



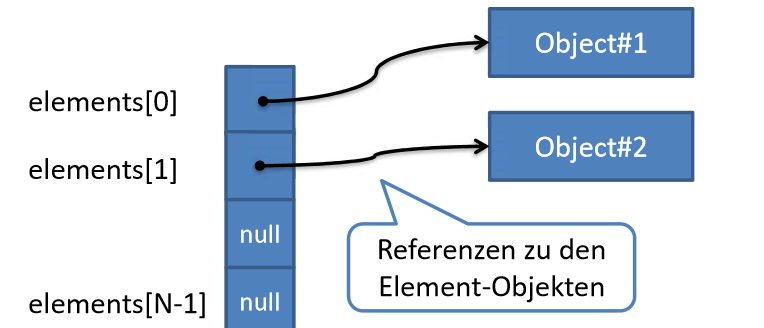
Das Auto hat einen Motor! Es ist selber keinen!

Kommentare

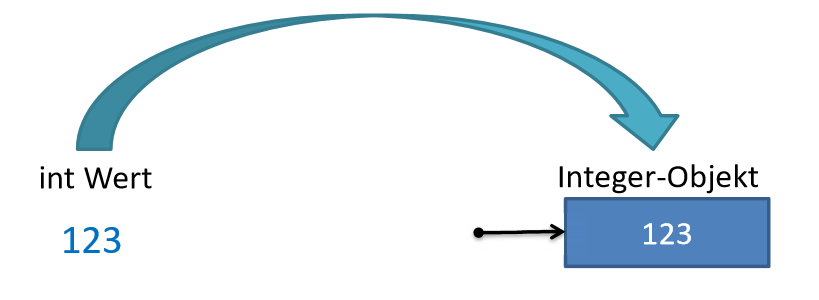
Kommentare sollten wenn möglich vermieden werden. Sie sind meist ein Zeichen für schlechten Code.

# Collections

Collections ist ein sehr wichtiges Thema, die Anwendung von Collections kommt bestimmt an der Prüfung.

Wrapper Klassen / Boxing

Auf einen Stack kann man Objekte legen und wieder wegnehmen(Push and pop FIFO-Verfahren). Auf dem Stack werden nur die Referenzen von Objekten gespeichert.

Da die Referenzen von Objekten im Stack gespeichert werden, können keine Wertetypen auf den Stack gelegt werden. Somit ist es uns nicht möglich, primitive Datentypen wie int, short, double, float oder char auf dem Stack abzulegen. Um trotzdem solche Werte auf den Stack zu legen, hat man Wrapper-Klassen eingeführt.  
Mit diesen Wrapper-Klassen ist es möglich aus einem primitiven Datentyp, beispielsweise int, einen Objekttyp(Integer) zu erstellen. Man erstellt also ein Objekt mit einer Referenz und einem Wert. Dann ist es möglich, den Wert per Referenz anzusprechen. Das ist die Lösung um auch primitive Datentypen auf den Stack zu speichern.

Für jeden primitiven Datentyp gibt es eine Wrapper-Klasse dazu:

Diese Wrapper-Objekte kann man nicht nur auf den Stack legen, sondern auch in Collections verwenden.

Unboxing

Das Verpacken in eine Wrapper Klasse nennt man Boxing, das Auspacken Unboxing. Diese Vorgänge können implizit und explizit geschehen:

**Implizites Boxing / Auto-Boxing**



**Aufpassen, dass es keine NullPointerException gibt**

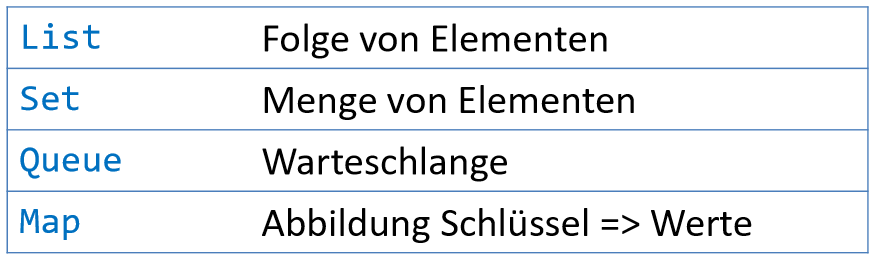
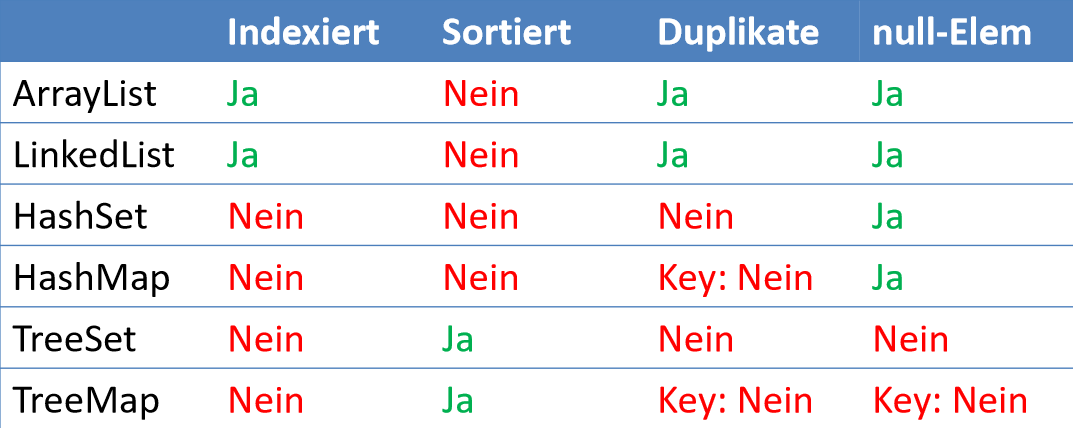
**Auto-Boxing Ausnahmen**

**Double** **d** = 4; //Funktioniert nicht weil int-

**Double** **d** = 4.0f; // Funktioniert nicht weil float

**Double** **d** = 4.0; // Funktioniert

Collections-Übersicht

Es gibt vier verschiedene Arten von Collections:

Array-List

Eine Array-Liste ist ein Array, welches sich selber erweitert und implementierte Funktionen hat.

**Funktionen / Befehle**

**ArrayList**<Bottle> **bottles** = **new** ArrayList<>(); // Erstellen einer Array-List

bottles.add(bottlePlastic); //Hinzufügen von Element an letzter Stelle  
bottles.add(1, bottleGlas); //Hinzufügen an Stelle x[0 bis n-1] alle Elemente werden eins nach hinten geschoben. Hinzufügen -> Liste wird um eins grösser.

bottles.remove(0); //Entfernen eines Elements aus der Array-Liste Index [0 bis n-1], Array-Liste wird um eins kleiner, alle Elemente eins nach vorne

bottles.set(0, bottlePlastic);**// Element an Stelle [Index] wird ersetzt durch neues Element.**

bottles.get(1) //Gibt Element der Stelle [Index] zurück

bottles.isEmpty() //Prüft ob die Arraylist leer ist

bottles.contains(bottleGlas) // Prüft ob das Element vorhanden ist

bottles.size() // Grösse der Arraylist

bottles.indexOf(bottleGlas) // Gibt den Index[0 bis n-1] des Elements zurück

* Duplikate & null möglich
* Remove nur mit Iterator möglich
  + Ansonsten ConcurrentModificationException

**Iterator / remove**

Iterator<Bottle> **iterator** = bottles.iterator();

**while** (iterator.hasNext())

{

**Bottle** **current** = iterator.next();

**if** (current.equals(bottleIron))

{

iterator.remove();

} **else**

{

**System**.***out***.println(current.getMaterial());

}

}

Mit dem Iterator kann man Elemente entfernen!

Linked-List

Eine Linked-List ist eine verkettete Liste und daher schneller beim Einfügen und Löschen. Sie hat aber die gleichen Befehle wie die ArrayList:

**LinkedList**<Bottle> **bottles** = **new** LinkedList<>();  
FIFO Queue  
bottles.addLast(bottleIron);  
bottles.removeFirst();  
LIFO  
bottles.addLast(bottleIron);  
bottles.removeLast();

Eine Linked-List kann man gut als Queue benutzen, da sie die obigen Befehle unterstützt.

Set-Übersicht

Ein Set zeichnet sich dadurch aus, dass es keine Duplikate zulässt. Es ist somit nicht möglich, dass ein Element mehrfach in einem Set vorkommt.

TreeSet

**Übersicht**

**Ein TreeSet hat folgende Eigenschaften:**

* **Sortiert**
* **Immer effizient**
* **Elemente Comparable implementiert**
* **Equals() implementiert**

**Funktionen / Befehle**

**TreeSet**<Bottle> **bottles** = **new** TreeSet<>(); // Erstellt ein TreeSet

bottles.add(bottleGlas); //Fügt ein Element hinzu

bottles.remove(bottlePlastic); //Entfernt ein Element

bottles.size(); // Grösse des Sets

bottles.isEmpty() //Prüft ob das Set leer ist

bottles.contains(bottleGlas) // Prüft ob das Element vorhanden ist

bottles.first() //Erstes Element(bemerke ist sortiert!)

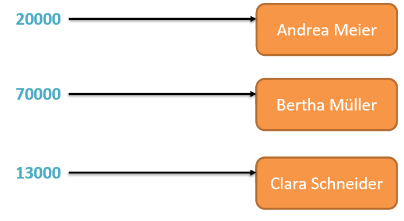
bottles.last() // Letztes Element

**HashSet**

* **Unsortiert**
* **Oft sehr effizient**
* **hashCode() implementiert**
* **equals() implementiert**

**HashSet**<Bottle> **bottles** = **new** HashSet<>(); // Erzeugt neues HashSet

Ansonsten die gleichen Befehle ausser first() und last().

Map-Übersicht

Eine Map beinhaltet immer einen Schlüssel und einen dazugehörigen Wert. Am einfachsten ist dies, wenn der Schlüssel eine Zahl is, wie wir es und gewohnt sind. Doch kann es auch jederzeit sein, dass die Schlüssel Objekt sind und die Werte ebenfalls aus Objekten bestehen.

**Beispiel**

Man hat verschiedene Flaschen und dazugehörige Herstellen(Marken) und will diese Ausgeben:

**Bottle** **bottlePlastic** = **new** Bottle(1.5, "plastic");

**Bottle** **bottleIron** = **new** Bottle(0.5, "iron");

**Bottle** **bottleGlas** = **new** Bottle(1.0, "glas");

**Brand** **firstBrand** = **new** Brand("adidas");

**Brand** **secondBrand** = **new** Brand("puma");

**Brand** **thirdBrand** = **new** Brand("sponser");

**TreeMap**<Bottle, Set<Brand>> **bottles** = **new** TreeMap<>();

**TreeSet**<Brand> **brandsPlastic** = **new** TreeSet<>();

**TreeSet**<Brand> **brandsIron** = **new** TreeSet<>();

**TreeSet**<Brand> **brandsGlas** = **new** TreeSet<>();

brandsPlastic.add(firstBrand);

brandsPlastic.add(secondBrand);

brandsIron.add(secondBrand);

brandsIron.add(thirdBrand);

brandsGlas.add(thirdBrand);

bottles.put(bottlePlastic, brandsPlastic);

bottles.put(bottleIron, brandsIron);

bottles.put(bottleGlas, brandsGlas);

**for** (Entry<Bottle, Set<Brand>> **entry** : bottles.entrySet())

{

**System**.***out***.println(entry.getKey().getMaterial() + ":");

**for** (**Brand** **brand** : entry.getValue())

{

**System**.***out***.println(brand.getName());

}

**System**.***out***.println("\_\_\_\_\_\_\_\_\_\_\_\_");

}

TreeMap

* Nach Schlüssel sortiert
* Immer effizient
* Comparable und equals() implementiert

**Funktionen / Befehle**

**TreeMap**<Bottle, Set<Brand>> **bottles** = **new** TreeMap<>(); //Erstellen von TreeMaps

bottles.put(bottlePlastic, brandsPlastic); //Element auf die Map legen

bottles.remove(bottleGlas); //Element aufgrund des Schlüssels entfernen  
bottles.remove(bottleGlas, brandsGlas); //Element aufgrund des Schlüssels und des Wertes entfernen

bottles.containsKey(bottleGlas)// Prüft ob der Key vorhanden ist

bottles.containsValue(brandsGlas) //Prüft ob der Inhalt vorhanden ist

bottles.size(); // Grösse der Map

bottles.isEmpty() //Prüft ob das Set leer ist

bottles.firstKey() // Schlüssel des ersten Elements

bottles.firstEntry() // Wert des ersten Elements

bottles.lastKey()// Schlüssel des letzten Elements

bottles.lastEntry() //Wert des letzten Elements

if(!hashmap.containsKey(key)) {

p.put(key,1);

}

else {

hashmap.put(key, hashmap.get(key)+1);

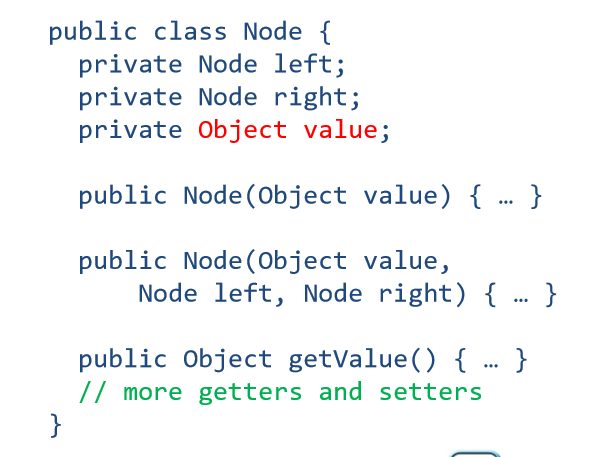
}

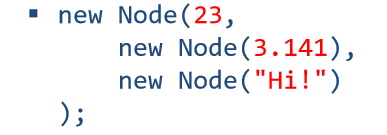
HashMap

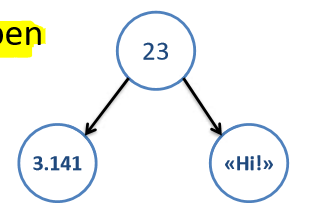
* **Unsortiert**
* **Oft sehr effizient**
* **hashCode() konsistent zu equals()**

# Generics

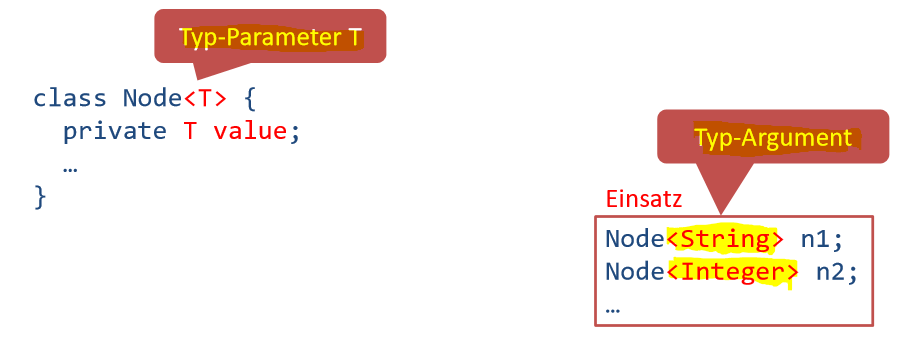
Mit Generics erzwingt man, dass man nur den gleichen Elementtyp brauchen kann. Man braucht Generics in Collections, Interfaces und Klassen.

Somit kann man mit Generics Homogenität erzwingen.

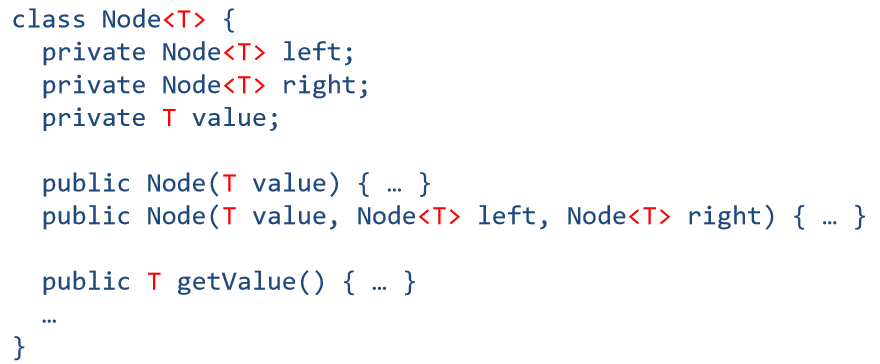




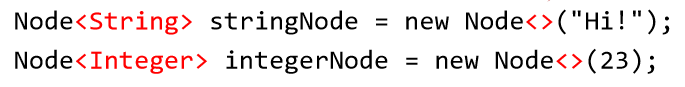
Generische Klasse

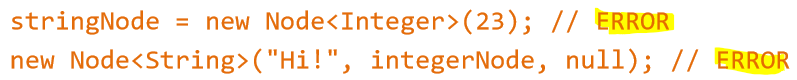
Eine Generische Klasse ist eine Klasse mit Typ-Parameter(<T>). Ein Typ-Parameter ist ein Platzhalter für einen unbekannten Daten-Typ. Gibt es eine Instanziierung, muss das Typ-Argument angegeben werden.

Typ-Variable

Eine Typ-Variabel ist der Typ-Parameter einer generischen Klasse. In der Klasse kann die Variabel T so verwendet werden, wie ein normaler Datentyp:

Generische Typen

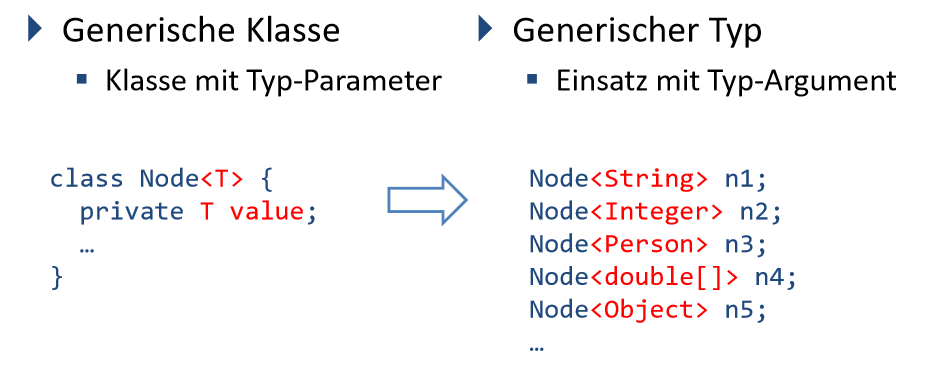
Bei der Instanziierung einer generischen Klasse muss ein generischer Typ angegeben werden. Der generische Typ besteht dabei aus gewöhnlichen Datentypen:

Eine Statische Typ-Prüfung funktioniert nicht:

BildschirmausschnittDas Casten entfällt:

Generische Typen werden zur Compile-Zeit geprüft, das garantiert:

* Korrekte Verwendung und Kompatibilität garantiert
* Typ-Fehler vom Compiler erkannt 🡪 keine Type-Laufzeit-Fehler mehr

Beispiel mit einem Typ-Parameter

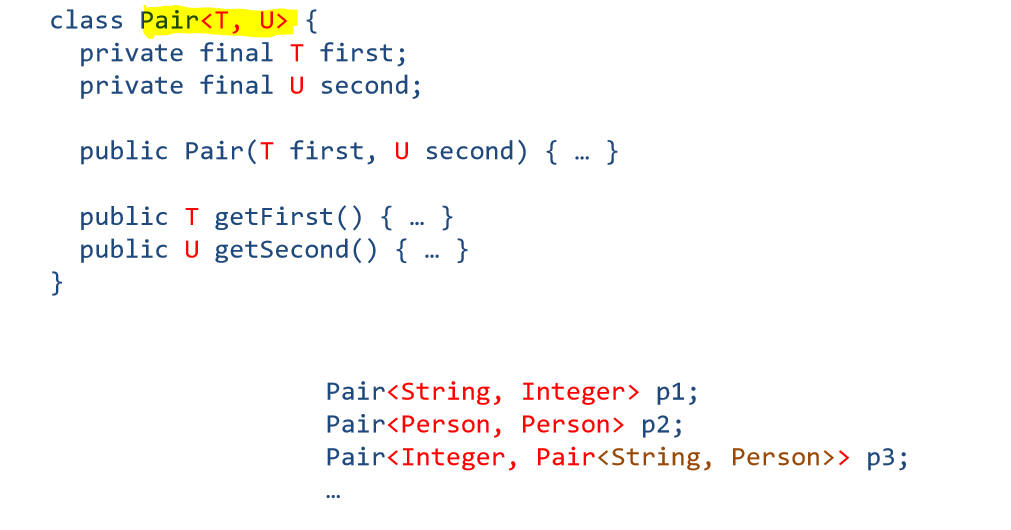
|  |  |
| --- | --- |
| **Klasse mit Typ-Parameter** | **Instanzierungsklasse** |
| **public** **class** **Node**<**T**>  {  **private** **Node**<**T**> last;  **private** **Node**<**T**> next;  **private** **T** value;    **public** **Node**(**T** value)  {  **this**.value = value;  }    **public** **Node**(**T** value, **Node**<**T**> last, **Node**<T> next)  {  **this**.value = value;  **this**.last = last;  **this**.next = next;  }  } | **Node**<Integer> **n1** = **new** Node<>(1);  **Node**<Integer> **n3** = **new** Node<>(2);  **Node**<Integer> **n2** = **new** Node<>(3, n1, n3);    **Node**<String> **n4** = **new** Node<>("four");  **Node**<String> **n6** = **new** Node<>("five");  **Node**<String> **n5** = **new** Node<>("six", n4, n6); |

Generische Interface

Es gibt auch generische Interfaces(bsp alle Collection oder Comparable):

Mehrere Typ-Parameter

Das beste Beispiel mit mehreren Typ-Parameter sind die Maps<T1, T2>.



|  |  |
| --- | --- |
| **Klasse mit Typ-Parameter** | **Instanzierungsklasse** |
| **public** **class** **Node**<**T**, **U**>  {  **private** **Node**<**T**, **U**> last;  **private** **Node**<**T**, **U**> next;  **private** **T** value;    **public** **Node**(**T** value)  {  **this**.value = value;  }    **public** **Node**(**T** value, **Node**<**T**, **U**> last, **Node**<**T**, **U**> next)  {  **this**.value = value;  **this**.last = last;  **this**.next = next;  }  } | **Node**<Integer, String> n1 = **new** Node<>(1);  **Node**<Integer, String> n3 = **new** Node<>(2);  **Node**<Integer, String> n2 = **new** Node<>(3, n1, n3);    **Node**<String, Integer> n4 = **new** Node<>("four");  **Node**<String, Integer> n6 = **new** Node<>("five");  **Node**<String, Integer> n5 = **new** Node<>("six", n4, n6); |

Type-Bound

Bis anhin durfte man alles für den Parameter T (&U) eingeben. Nun wollten wir dies vermeiden und feststellen, dass die Typ-Argumente von einer Klasse abhängen:

**public** **class** **Node**<**T** **extends** **Person**> {…}

**Node**<Person> **personNode**; //OK

**Node**<Student> **studentNode**; // OK

**Node**<String> **stringNode**; //Fehler

TreeSet / Comparable

Damit Nodes in einem TreeSet gespeichert werden können, müssen diese Comparable sein:

**public** **class** **Node**<**T** **extends** Comparable<**T**>>{…}

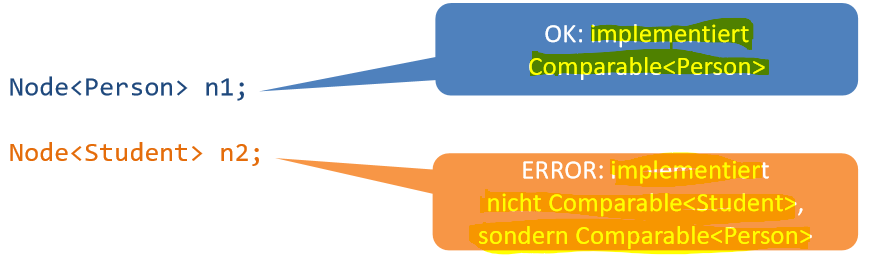
**public** **class** **Person** **implements** Comparable<Person>{…}

**Node**<Person> **personNode**; //ok 🡪 Comparable<Person>

**Node**<Student> **studentNode**; Fehler 🡪Comparable<Student>

**Node**<String> **stringNode**;🡪OK Comparable<String>

{



Mehrere Type Bounds

Es ist möglich, dass man mehrere Type Bounds einsetzten kann:

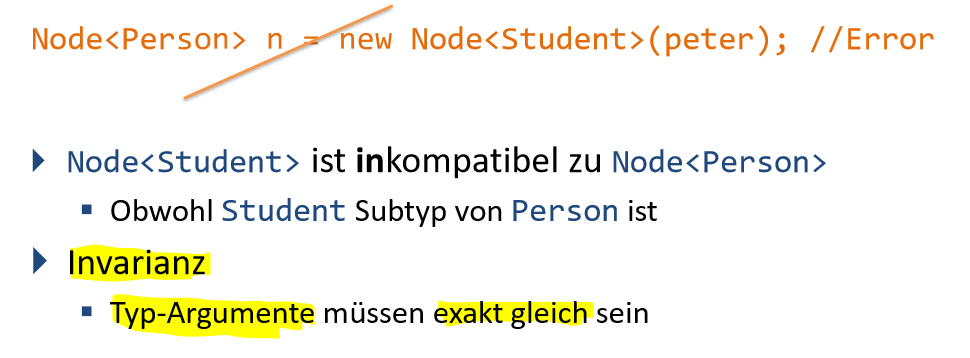
Bildschirmausschnitt  
Bildschirmausschnitt

Generische Methoden

Generische Methoden verwenden Typ-Variablen:

|  |  |
| --- | --- |
| **Generische Methode** | **Aufruf(Typ-Argument automatisch erkennt)** |
| **public** **class** **Test**<**T**>  {  **private** **T** x;  **private** **T** y;  **private** **T** z;  **public** **Test**(**T** x, **T** y, **T** z)  {  **this**.x = x;  **this**.y = y;  **this**.z = z;  }  **public** <**T**> **T** **majority**(**T** x, **T** y, **T** z)  {  **if** (**this**.x.equals(y))  {  **return** x;  }  **if** (**this**.x.equals(z))  {  **return** y;  }  **if** (**this**.y.equals(z))  {  **return** z;  }  **return** **null**;  }  } | **Test**<Integer> **testOne** = **new** Test<>(1, 2, 3);    **System**.***out***.println(testOne.majority(0, 5, 2));    **Test**<String> **testTwo** = **new** Test<>("eins", "zwei", "drei");  **System**.***out***.println(testTwo.majority("null", "sieben", "zwei")); |

Invarianz



Wildcard-Typ

Um der Invarianz aus dem Weg zu gehen, gibt es Wildcard-Typen:

**Node**<?> **undefinedNode**;

undefinedNode = **new** Node<Integer>(4);  
undefinedNode = **new** Node<String>("four");

Mit dem Wildcard zeigt man an, dass man noch nicht wiess, welchen Typ man erwartet. Jedoch kann man nicht lesen und schreiben.

**Covarianz**

**Ebenfalls kann man sagen, dass man nur weiss, dass es von einer gewissen Klasse abhängen muss:**

**Node**<? **extends** **Person**> **undefinedNode**;  
undefinedNode = **new** Node<String>("four");//unzulässig  
undefinedNode = **new** Node<Student>(); //Zulässig

**Lesen erlaubt, Schreiben verboten.**

**Contravarianz**

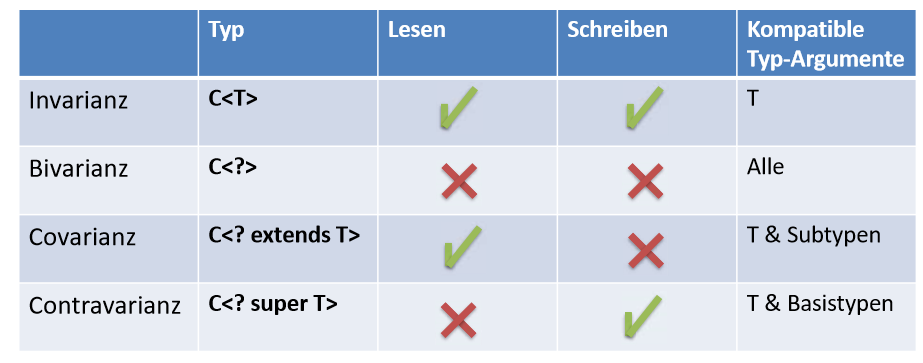
**Ebenfalls kann man das Gegenteil sagen: Es muss eine Basis-Klasse oder Super-Klasse sein:**

**Node**<? **super** **Person**> **undefinedNode**;

undefinedNode = **new** Node<Student>(); //Unmöglich

undefinedNode = **new** Node<Object>(); //Möglich

Schreiben erlaubt, Lesen verboten.

**Zusammenfassung**

# Lambdas

Praktische Collections Methoden

* Sort
* Max
* Min

**Collections**.*max*(bottles);

Bis anhin war die Funktion sortieren von der compareTo-Methode abhängig. Das heisst die Klasse hatte Comparable implementiert. Mittels Lambdas soll es möglich sein, selber Suchkriterien zu kreieren.

Methodenreferenz

Mittels Methodenreferenz teilt man einer Methode mit, welche die Kriterien sind:

**public** **void** **printSortedBySize**(**ArrayList**<Test> testList)

{

testList.sort(**this**::comparaBySize);

**for** (**Test** **test** : testList)

{

**System**.***out***.println(test.name + " " +test.size);

}

}

**public** **void** **printSortedByName**(**ArrayList**<Test> testList)

{

testList.sort(**this**::compareByName);

**for** (**Test** **test** : testList)

{

**System**.***out***.println(test.name + " " +test.size);

}

}

**public** **int** **comparaBySize**(**Test** t1, **Test** t2)

{

**return** **Integer**.*compare*(t1.size, t2.size);

}

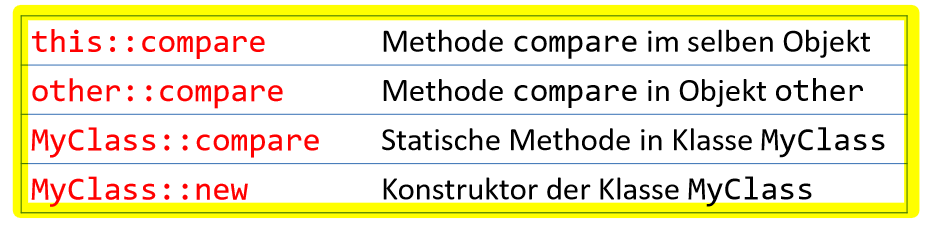
**public** **int** **compareByName**(**Test** t1, **Test** t2)

{

**return** t1.name.compareTo(t2.name);

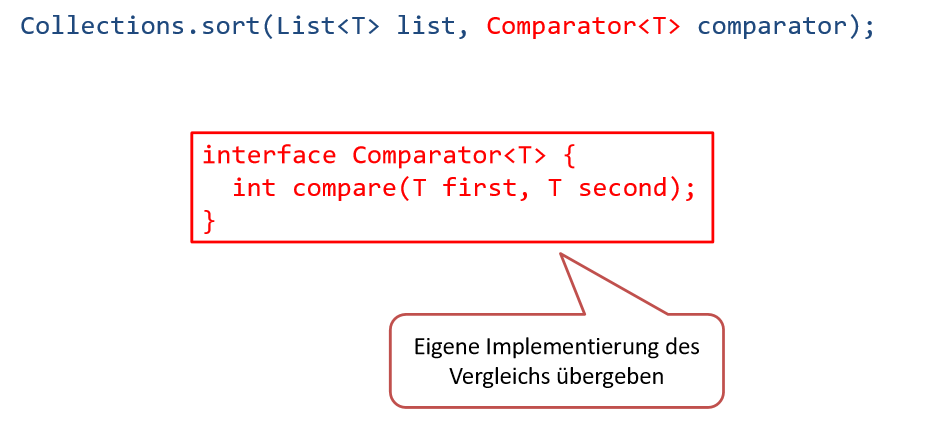
}

Es wird jeweils die Methode als Referenz mitgegeben.

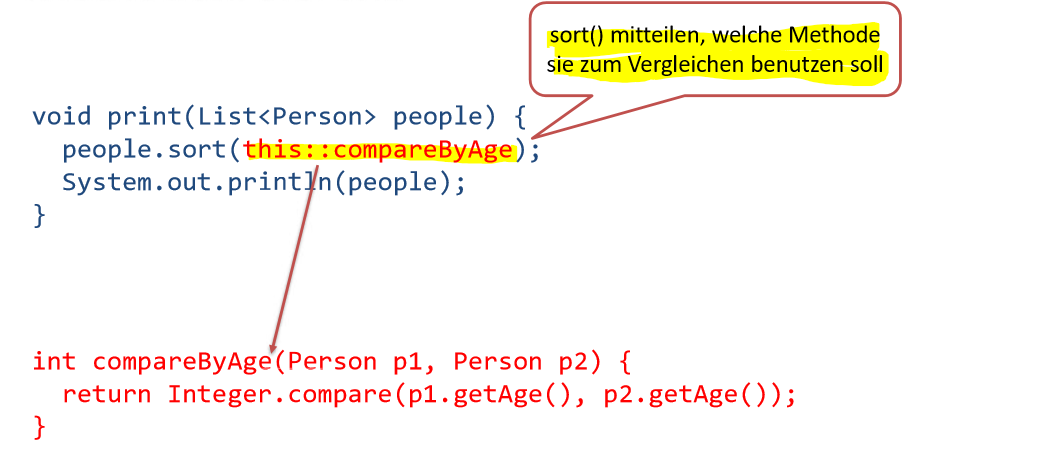


Funktionsschnittstellen

Die Sortier-Methode der Collections-Klasse beinhaltet eine Liste und ein Interface mit den Bedingungen nach welchem Kriterium sortiert werden soll:



Nun erwartet die Sortiermethode 2 Argumente:

1. Eine Liste
2. Eine Vergleichsmethode, welche gleich aufgebaut ist
   1. Dazu mehr bei den Höherwertigen Funktionen

Höherwertige Funktionen

Höherwertige Funktionen sind Funktionen, bei welchen die Logik nicht vollständig implementiert ist. Stattdessen erwarten solche Methoden Vergleichsmethoden(z.B mittels Merthodenreferenz) als Parameter.   
Das beste Beispiel ist die Sort-Methode. Diese braucht als Parameter eine Vergleichsmethode etwa: this:comparyByAge.

Die Parameter, welche als Argument eine Vergleichsmethode braucht, werden als Funktionale Schnittstellen definiert.

Diese Funktionale Schnittstelle hat eine abstrakte Methode.

Die Signatur(Parameter) und der Rückgabetyp der Vergleichsmethode muss zur abstrakten Methode des Interfaces passen.

|  |  |
| --- | --- |
| **Funktionale Schnittstelle(java.Comparator)** | **Methode der auf welche referenziert wird** |
| ***@FunctionalInterface***  **public** **interface** FilterCriterion<**T**>  {  **public** **abstract** **i**nt [compare](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#compare-T-T-)([T](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) o1, [T](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) o2)  } | **public** **static** **int** **compareBySize**(**Test** t1, **Test** t2)  {  **return** **Integer**.*compare*(t1.getSize(), t2.getSize());  } |

Es ist wichtig, dass das Interface und die Methode den gleichen Rückgabewert und die gleichen Parameter haben.

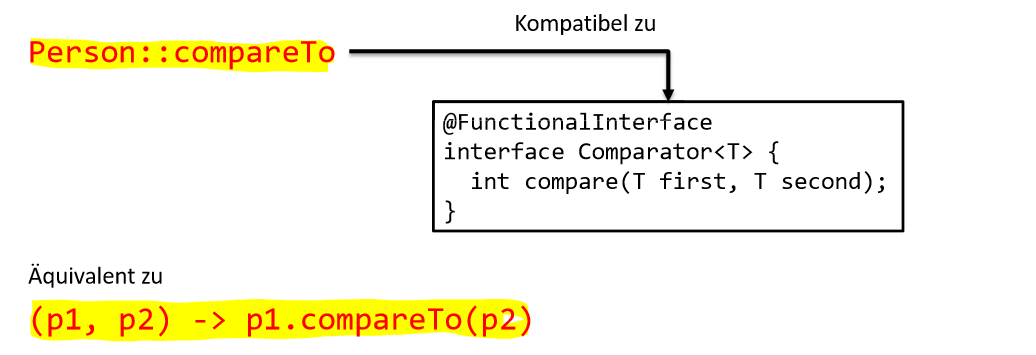
Lambda

Damit man keine unnötigen Hilfsmethoden mehr hat, ist es mittels Lambdas möglich, die Logik direkt zu implementieren:

Parametertypen können weggelassen werden, diese werden automatisch erkennt. Ebenfalls braucht es die geschweiften Klammern bei einem logischen Ausdruck nicht:  
Bildschirmausschnitt

|  |  |
| --- | --- |
| **Funktionale Schnirttstelle(java.Comparator)** | **Lambda bySize)** |
| ***@FunctionalInterface***  **public** **interface** FilterCriterion<**T**>  {  **public** **abstract** **i**nt [compare](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#compare-T-T-)([T](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) o1, [T](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) o2)  } | testList.sort((**t1**, **t2**) -> **Integer**.*compare*(t1.getSize(), t2.getSize())); |

|  |  |
| --- | --- |
| **Funktionale Schnirttstelle(java.Comparator)** | **Lambda(byName)** |
| ***@FunctionalInterface***  **public** **interface** FilterCriterion<**T**>  {  **public** **abstract** **i**nt [compare](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#compare-T-T-)([T](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) o1, [T](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) o2)  } | testList.sort((**t1**, **t2**) -> t1.getName().compareTo(t2.getName())); |



**Vorgehensweise andere Lambda-Methoden(nicht sort)**

|  |  |
| --- | --- |
| **Funktionale Schnirttstelle(selbstgemacht)** | **Aufruf(Typ-Argument automatisch erkennt)** |
| ***@FunctionalInterface***  **public** **interface** FilterCriterion<**T**>  {  **boolean** **test**(**T** t1);  } | *filter*(testList, **t1** -> t1.getSize() >= 4); |

**public** **static** <**T**> Collection<**T**> **filter**(Collection<**T**> input, FilterCriterion<**T**> criterion)

{

Iterator<**T**> **it** = input.iterator();

**while**(it.hasNext())

{

**if**(criterion.test(it.next())) {

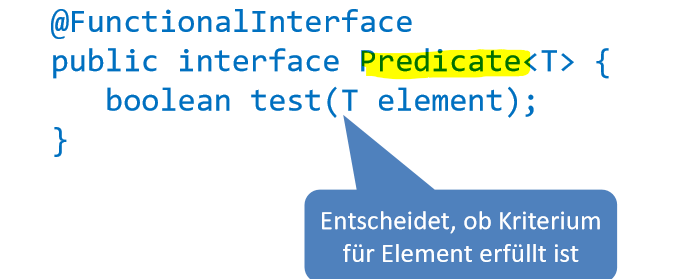
it.remove();

}

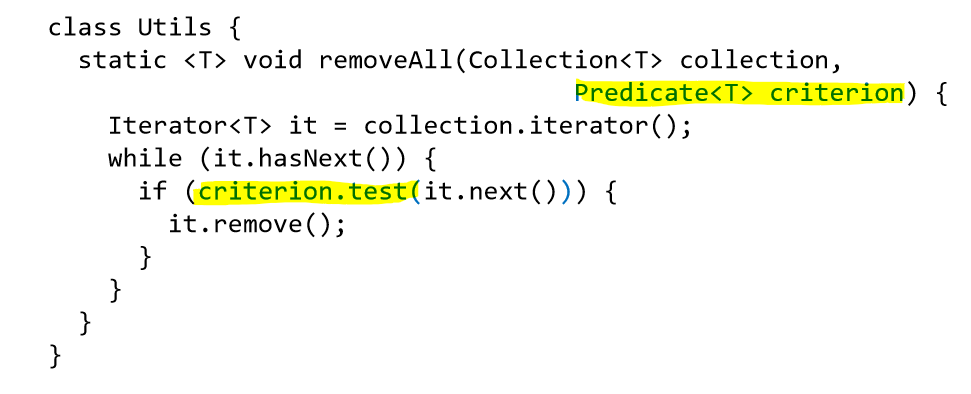
}

**return** input;

}

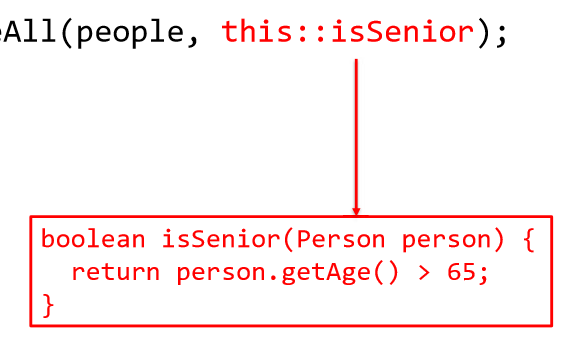


1. Funktionsschnittstelle für Kriterium definieren
   1. Abstracte Methode
2. Funktion schreiben/einbinden(z.B filter)



1. Filter aufrufen mit Lambda(oder Methodenreferenz)

Bildschirmausschnitt



Comparator Bausteine

Ebenfalls kann man für die Sortierung auch Comparator-Bausteine verwenden:

testList.sort(Comparator.*comparing*(Test::getSize)); //By Size

testList.sort(Comparator.*comparing*(Test::getName).thenComparing(Test::getSize).reversed());  
// Zuerst nach Name, danach nach umgekehrter

Closure

Closure nennt man die Nutzung von Lokalen Variablen der umgebenden Methode(hier filterCriterion):

**System**.***out***.println("Print Criterion");

**Scanner** **scanner** = **new** Scanner(**System**.***in***);

**String** **filterCriterion** = scanner.nextLine();

*filter*(testList, **t1** -> t1.getName().contains(filterCriterion));

Variablen, auf welche in den Lambdas zugegriffen sind, sind automatisch final.

Geschachtelte Klassen

In Woche 10 haben wir ebenfalls noch geschachtelte Klassen kennengelernt. Auf diese gehe ich nicht ein, da dies nur für seltene Projekte relevant ist.

# Stream API

Die Stream-API ist ein Framework für die Vereinfachung von Collection-Operatoren. Sie beinhaltet Lambdas oder Methodenreferenzen als Argumente:

**Anwendung**

testList.stream()

.filter(**t** -> !(t.getName().contains(filterCriterion))) //Filterung, alles was NICHT = filterCriterion ist bleibt!(where getname not like …)

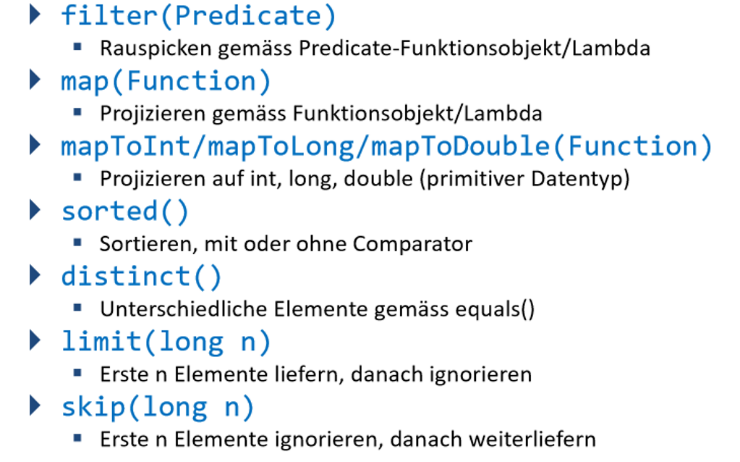
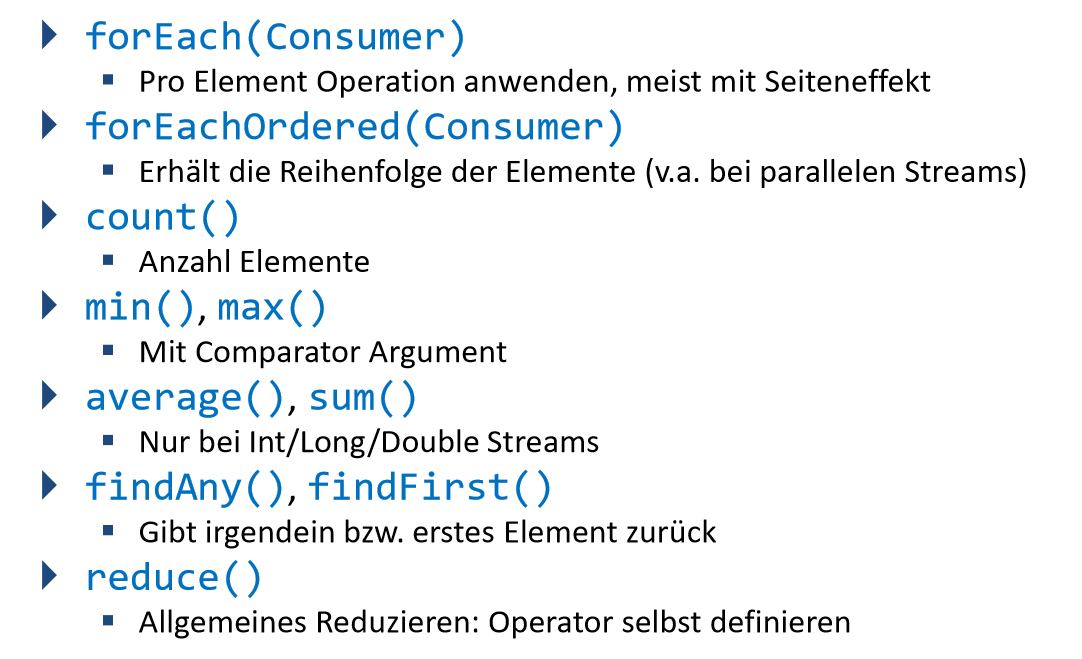
.sorted(Comparator.*comparing*(Test::getSize).reversed()) //Sortierung mit/ohne Comparator

.map(**t** -> t.getName() +" " + t.getSize()) //Wie ist soll die Ausgabe sein

.limit(2) // limitiert die Ausgabe auf 2 Datensätze

.skip(1) // Überspringt ein Datensatz

.forEach(**System**.***out***::println);//ausgabe für jedes Element



**Wichtige Funktionen**

**Average, Sum, Min, Max**

**OptionalDouble** **avgSize** = testList.stream().mapToInt(**t** -> t.getSize()).average();

avgSize.ifPresent(**System**.***out***::println);

oder

testList.stream()

.mapToDouble(**t** -> t.getSize()).average()

.ifPresent(**System**.***out***::println);

**int** **sumSize** = testList.stream().mapToInt(**t** -> t.getSize()).sum();

**System**.***out***.println(sumSize);

Oder

**System**.***out***.println(testList.stream()

.mapToInt(**t** -> t.getSize()).sum());

testList.stream().mapToInt(**t** -> t.getSize()).min()

.ifPresent(**System**.***out***::println);

oder

**OptionalInt** **maxSize** = testList.stream()

.mapToInt(**t** -> t.getSize()).max();

**System**.***out***.println(maxSize);

**Reduce(Verknüpfen)**

**System**.***out***.println(testList.stream().map(**t** -> t.getName()).reduce("Strings: ", (**t1**, **t2**) -> t1 + " " + t2));

oder

**String** **testNames** = testList.stream()

.map(**t** -> t.getName())

.reduce("Names:", (**t1**, **t2**) -> t1 +" " +t2);

**System**.***out***.println(testNames);

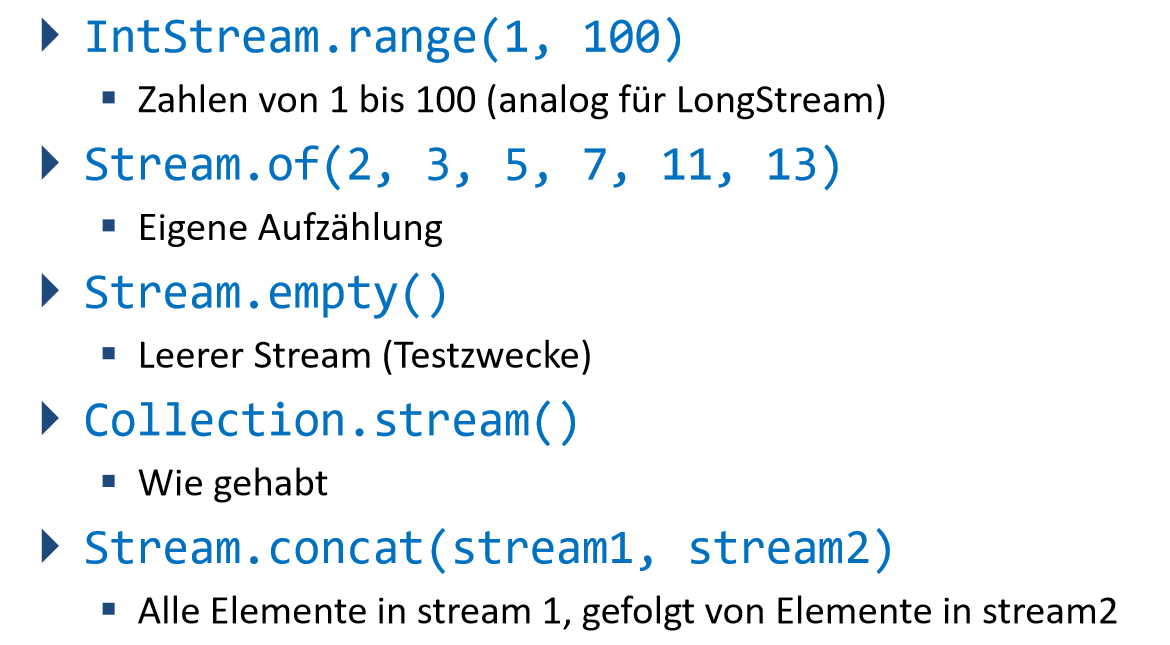
**allMatch(), anyMatch(), noneMatch()**

Prüft ob keines, eines oder alle auf eine Bedingung passen.

**Endliche Quellen**

IntStream.*range*(1, 100)

.forEach(**System**.***out***::println);

****

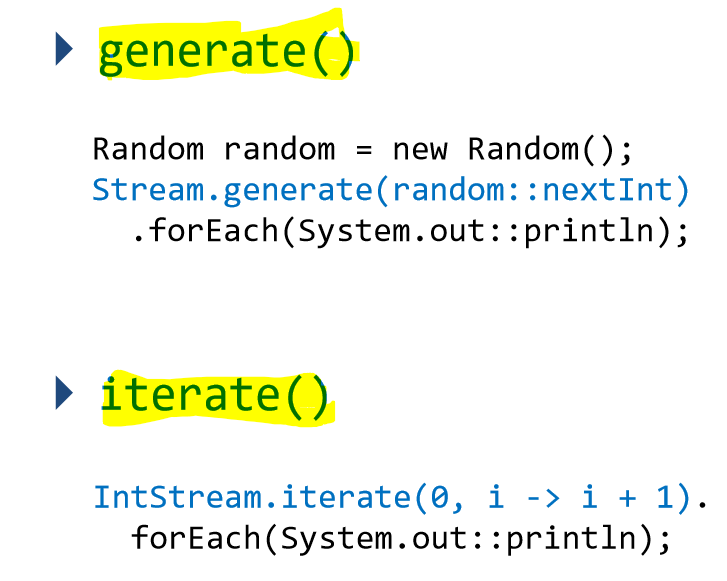
**Unendliche Quellen**

**Random** **random** = **new** Random();

IntStream.*generate*(random::nextInt)

.limit(100)

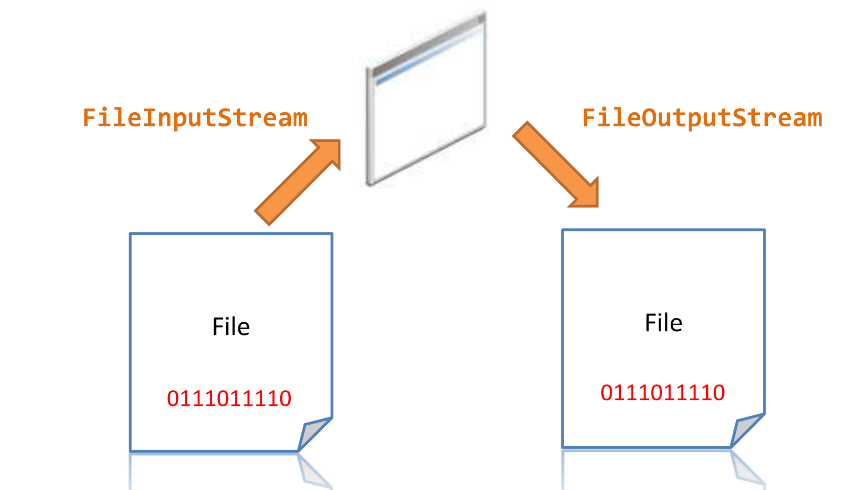
.forEach(**System**.***out***::println);

****

# Input / Output

Mittels Streams wollen wir von Datenquellen lesen oder in diese schreiben. Diese Streams sind nicht zu vergleichen mit der Stream-API.

Byte-Streams

Byte-Streams sind «Ströme» von Bits, also Ströme von Nullen und Einsen.

Somit kann man aus einer Datei lesen und schreiben, empfiehl sich, wenn man Zahlen lesen und bearbeiten will. Beispielsweise macht das folgende Programm aus «1234» «2345»:

**try** (**FileInputStream** **in** = **new** FileInputStream("test.data"))

{

**try** (**FileOutputStream** **out** = **new** FileOutputStream("test\_write.data"))

{

**int** **value** = in.read();

**while** (value >= 0)

{

**byte** **b** = (**byte**) value;

b++;

out.write(b);

**System**.***out***.println(b);

value = in.read();

}

} **catch** (**IOException** **e**)

{

**System**.***out***.println("IOExeption: " + e);

}

} **catch** (**IOException** **e**)

{

**System**.***out***.println("other Exception: " + e);

}

Um die Systemaufrufe zu minimieren, kann man auch immer mehrere bytes lesen:

**byte**[] **buffer** = **new** **byte**[4];

**int** **bufferLength** = in.read(buffer);

**while** (bufferLength >= 0){

**byte** **b** = (**byte**) bufferLength;

**for**(**int** **i** = 0 ; i< buffer.length; i++){

buffer[i] += 1;

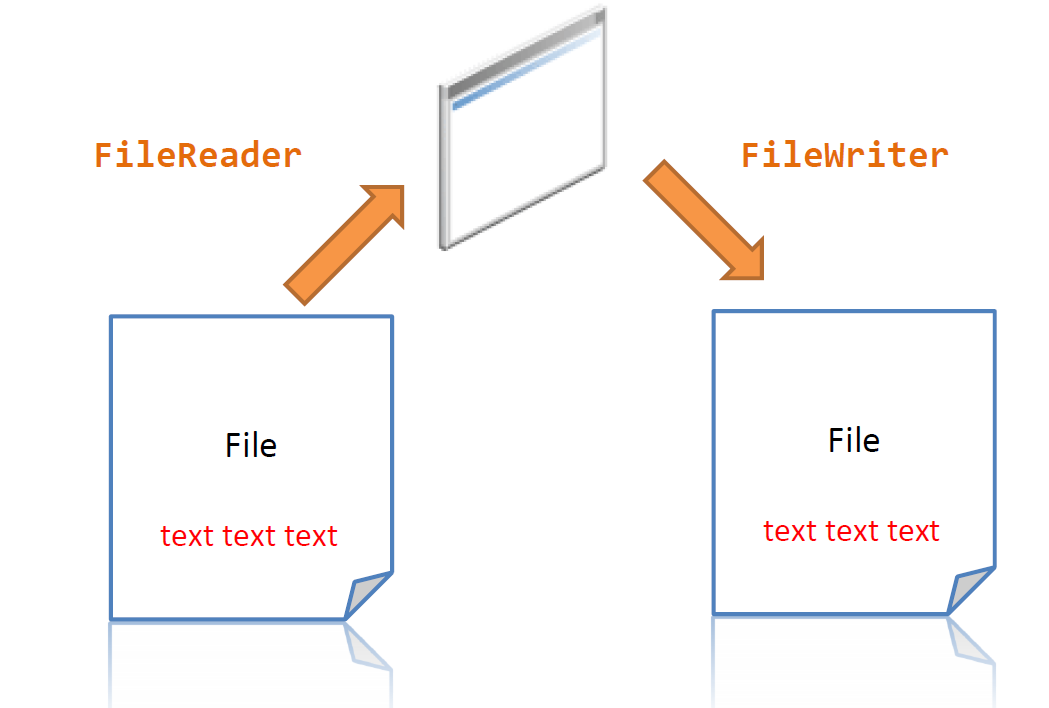
}

out.write(buffer, 0, bufferLength);

bufferLength = in.read();

}

Character Streams

Character Strams braucht man um Text einzulesen, respektive zu schreiben:  
**try** (**FileReader** **reader** = **new** FileReader("test.txt"))

{

**try** (**FileWriter** **writer** = **new** FileWriter("test\_write.txt"))

{

**int** **value** = reader.read();

**while** (value >= 0)

{

**char** **c** = (**char**) value;

**if** (*testUmlaut*(c))

{

writer.write(*convertUmlaut*(c));

}

**else**

{

writer.write(c);

}

value = reader.read();

}

} **catch** (**IOException** **e**)

{

**System**.***out***.println("IOExeption: " + e);

}

} **catch** (**IOException** **e**)

{

**System**.***out***.println("other Exception: " + e);

}

Mit dem Writer kann man natürlich auch Strings schreiben/anhängen an eine Datei:

**try** (**FileWriter** **writer** = **new** FileWriter("test\_write.txt", **true**))

{

writer.write(" Angehängt!")

BufferedReader / BufferedStreams

Um weniger Lese- und Schreibzugriffe auf ein System zu haben, kann man mittels BufferedReader(aber auch BufferedStreams) arbeiten. Ebenfalls hat man einen Performance Gewinn, da die Daten gecachet werden:

**BufferedReader** **buffReader** = **new** BufferedReader(reader);

**String** **line** = buffReader.readLine();

**System**.***out***.println(line);

Der grosse Vorteil dabei ist, dass man ganze Linien einlesen und schreiben kann.

**BufferedReader** **buffReader** = **new** BufferedReader(reader);

**try** (**FileWriter** **writer** = **new** FileWriter("filewe.txt"))

{

**String** **line** = buffReader.readLine();

**while** (line != **null**)

{

writer.write(line);

**System**.***out***.println(line);

line = buffReader.readLine();

}

Bridge Klassen

Mittels Bridge Klassen kann man Character in Bytes und umgekehrt umwandeln:

**InputStream** **byteStream** = **new** FileInputStream("in.txt");

**Reader** **reader** = **new** InputStreamReader( byteStream, "UTF-8");

# Objekt Serialisierung

Mittels Objekt Serialisierung kann man Objekte ex- und wieder importieren:

**class** **Person** **implements** **Serializable** {

**private** **String** firstName;

**private** **transient** **int** age;

**private** (…) serialVersionUID = 1L;

}

Klassen müssen das Interface «Serializable» eingebunden haben.

Person person = **new** Person(...);

**try** (**ObjectOutputStream** outputStream = **new** ObjectOutputStream(**new** FileOutputStream("serial.bin")))

{

outputStream.writeObject(person);

}

**try** (**ObjectInputStream** inputStream = **new** ObjectInputStream(**new** FileInputStream("serial.bin"))) {

**Person** **p** = (**Person**) inputStream.readObject();

}

}

# Regex

Mittels Regex ist es möglich eine Textstruktur zu analysieren zu zerlegen und anhand gewisser Regeln zu verändern.

**Pattern** **pattern** = **Pattern**.*compile*("(Mo|Di|Mi|Do|Fr)\\s(0?[0-9]|1[0-9]|2[0-3]):([0-5][0-9])");

**Matcher** **matcher** = pattern.matcher(input);

**if** (matcher.matches())

{

**String** **workday** = matcher.group(1);

**switch** (workday)

{

**case** "Mo":

**System**.***out***.println("Monday" + " " + matcher.group(2) + ":" + matcher.group(3));

**break**;

**case** "Di":

**System**.***out***.println("Tuesday" + " " + matcher.group(2) + ":" + matcher.group(3));

**break**;

**case** "Mi":

**System**.***out***.println("Wedensday" + " " + matcher.group(2) + ":" + matcher.group(3));

**break**;

**case** "Do":

**System**.***out***.println("Thursday" + " " + matcher.group(2) + ":" + matcher.group(3));

**break**;

**case** "Fr":

**System**.***out***.println("Friday" + " " + matcher.group(2) + ":" + matcher.group(3));

**break**;

**default**:

**System**.***out***.println("No Workday!");

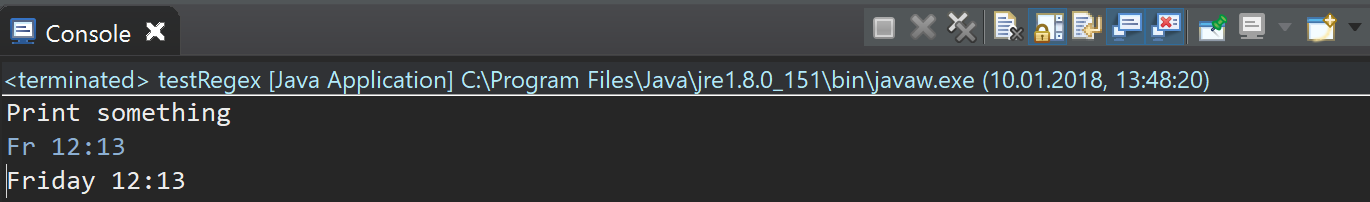
**break**;

}

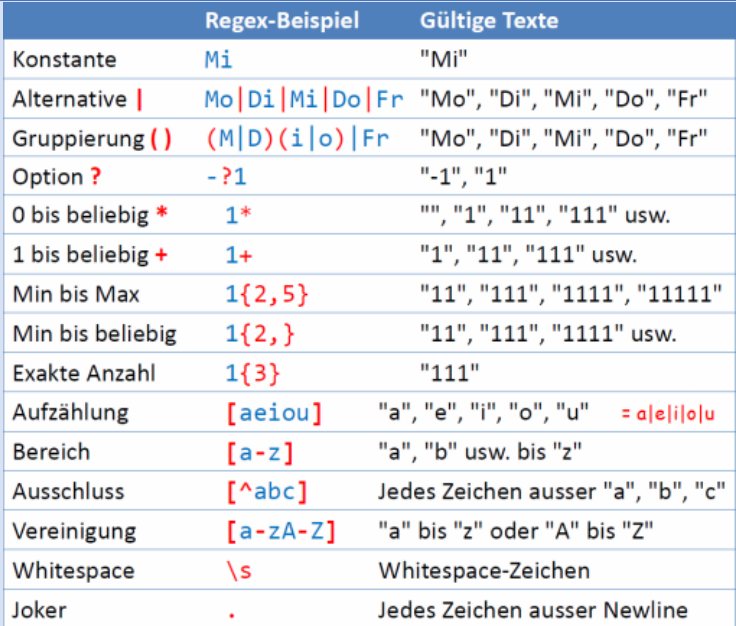
}

**else**

{**System**.***out***.println("No pattern matches!");}



Übersicht

}

Sonderzeichen

Sonderzeichen «( ) [ ] { } \* + ? | \» werden mit «\<Sondernzeichen> escapt:

\\ oder \(

# Unit Testing

Mit dem Unit Testing will man ein Programm systematisch testen. Der Hintergrund dabei ist «ein Programm testet ein Programm». Es soll also ein Code geschrieben, welche systematisch und automaisiert möglichst alle, relevante und spezielle, Fälle abdeckt.

Wir haben das Java Test Framework JUnit 5 dazu verwendet:

Relevante Testfälle

* Fullstack(Overflow)
* Pop on empty Stack(Underflow)
* LIFO(Last in First Out)
* New Stack(-1)
* Single push/pop
* NullPointerException
* Funktioniert Alles wie es soll
* Exception Testing, etc.

JUnit 5

* Pro Klasse eine Testklasse
* Pro Testfall eine Test-Methode
* Assert()-Methoden

Anwendung

1. Zu testende Klasse:

**public** **Wallet**(**int** startBalance)

{

balance = startBalance;

}

**public** **void** **deposit**(**int** amount ) **throws** **IllegalArgumentException**

{

**if**(amount <= 0)

{

**throw** **new** IllegalArgumentException("amount invalid ");

}

**else**

{

balance += amount;

}

1. JUnit-Testklasse:

**class** **TestWallet**

{

***@Test***

***@DisplayName***("Test Transfer negative amount")

**void** **testDepositNegativeAmount**() **throws** **IllegalArgumentException**

{

**Wallet** **wallet** = **new** Wallet(0);

*assertThrows*(**IllegalArgumentException**.**class**, () -> wallet.deposit(-1));

}

Assert-Methode

**Equals:**

***@Test***

***@DisplayName***("Test Wallet Creation all good")

**void** **testWalletCreationAllGood**()

{

**int** **amount** = 0;

**Wallet** **wallet** = **new** Wallet(amount);

*assertEquals*(amount, wallet.getBalance());

}

**Timeout**

***@Test***

***@DisplayName***("Test Timeout")

**void** **testWalletTimeot**()

{

**int** **amount** = 0;

**Wallet** **wallet** = **new** Wallet(amount);

*assertTimeoutPreemptively*(**Duration**.*ofSeconds*(2), () -> wallet.timeout());

}

**True / False**

***@Test***

***@DisplayName***("Test balance is Null")

**void** **testWalletBalanceIsNull**()

{

**Wallet** **wallet** = **new** Wallet(0);

*assertTrue*(wallet.getBalance() == 0);

wallet.deposit(100);

*assertFalse*(wallet.getBalance() == 0);

}

@BeforeEach, @AfterEach, @BerforeAll, @AfterAll

Mit den oben genannten Tags kann man erzwingen, dass etwas vor/nach jeder/allen Methoden ausgeführt wird. Ansonsten werden die Testmethoden nicht von oben nach unten ausgeführt sondern nach einer beliebigen Reihenfolge!

***@BeforeAll***

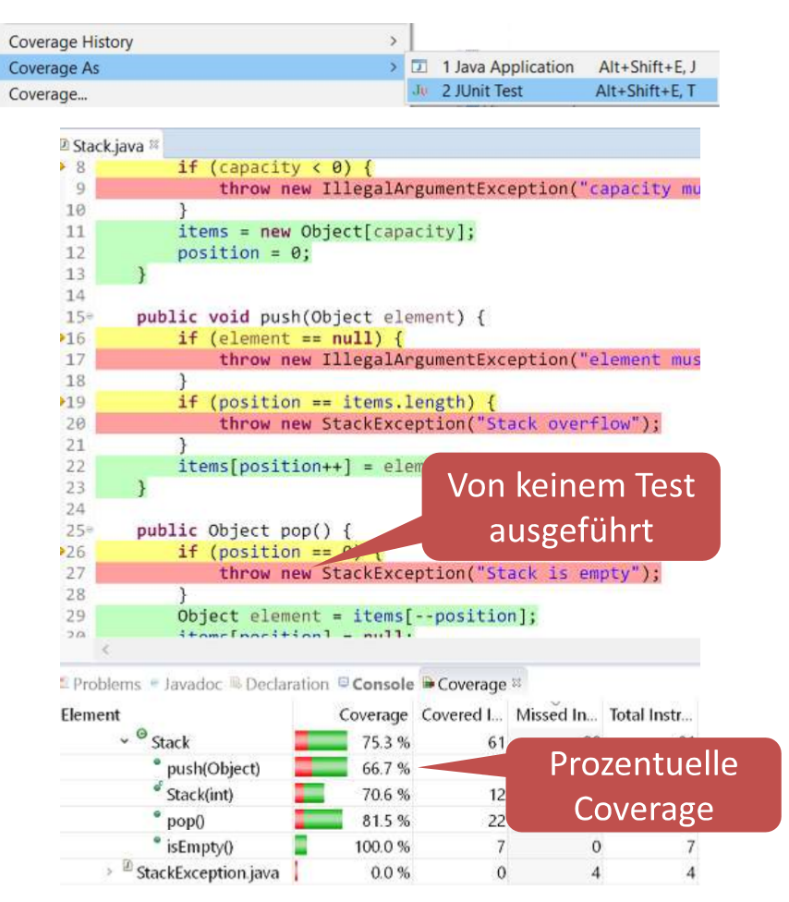
**public** **static** **void** **init**(){

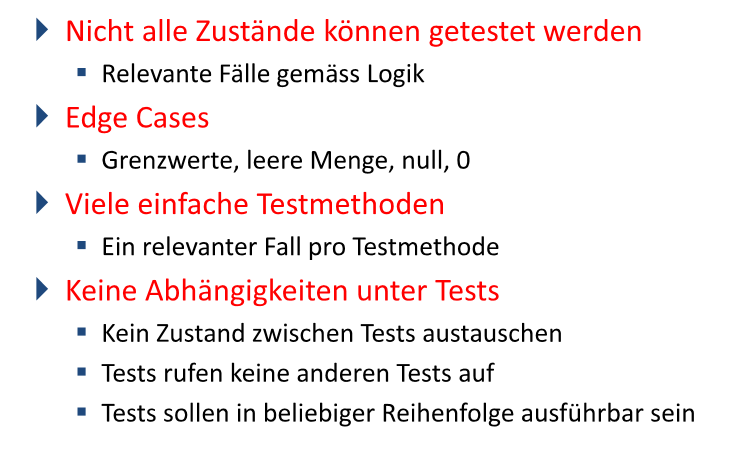
**System**.***out***.println("Before All init() method called");

}

Die @AfterAll und @BeforeALl Methoden müssen static sein.

Test Covarage

Die TestCoverafe zeigt die Abdeckung der Instruktionen & Pgade des Programmcodes durch Tests. Es ist eine gute Möglichkeit um zu schauen, was alles getestet wurde und wie viel Mal es aufgerufen wurde.  
Jedoch besagt die Coverage nur, wie viel Code das ungetestet wurde. Das bedeutet, eine tiefe Coverage zeigt, dass noch ungetesteter Code vorhanden ist. Eine Hohe Covarage sagt nur aus, dass viel Code getestet ist. Über die Korrektheit sagt die Covarage nichts aus. Eine 100% Covarage garantiert also noch keine Korrektheit.

Tipps fürs Testen