**Theorie**

*Ausdrücke*

|  |  |
| --- | --- |
| 1 + 2 \* 3 | 7 int |
| -3.0 / 2 | -1.5 double |
| 2 / 0 | ArithmeticException / by Zero |
| 3/0.0 | Double.POSITIVE\_INFINITY |
| "ABC".equals("abc") | true |
| '9' - '0' | 9 int |
| int a = 0, b = 0, c = 0;  boolean d = (a++ > 0 && b-- < 0) || --c < 0; | a = 1, b = 0, c= -1 & d = true |
| a + 1 < a | True 🡪a = Integer.MAX\_VALUE |

*Vergleiche*

|  |  |
| --- | --- |
| Gleitkommazahlen (float, double) | **Numerische** **Fehler** wegen der Maschinenpräzision werden bei Gleitkommazahlen nicht berücksichtigt, z.B. 3 \* 0.1 != 0.3. |
| String / Wrapper Types(Boolean, Character, Byte, Short, Integer) | Zwei gleiche Texte können in verschiedenen String- Objekten gespeichert sein und **== prüft nur Referenzgleichheit**, z.B. "A" != new String("A"). |

*Overloading, Overriding, Dynamic Dispatch*

|  |  |
| --- | --- |
| class Base {  void copyTo(Object other) {  // Method 1  }  void copyTo(Base other) {  // Method 2 | class Sub extends Base {  void copyTo(Base other) {  // Method 3  }  void copyTo(Sub other) {  // Method 4 |

|  |  |
| --- | --- |
| **überladene-Methoden**:  copyTo(Object) geerbt, copyTo(Base), copyTo(Sub) | b.copyTo(o); Methode 4  b.copyTo(b); Methode 3  b.copyTo(s); Methode 3 |
| **überschriebene Methoden:**  copyTo(Base) | s.copyTo(o); Methode 1  s.copyTo(b); Methode 3  s.copyTo(s); Methode 4 |
| ***1. Compiler: Overloading statischer(Methoden anhand Signatur)***  ***2. Laufzeitsystem: Ist die gewählte Methode überschrieben?(Dynamic Dipatch) 🡪 Methode anhand des dynamischen Types*** | |

*Funktionsschnittstellen*

|  |  |
| --- | --- |
| **Funktionale Schnirttstelle** | **Lambda** |
| ***@FunctionalInterface***  **interface** Comparision<**T,U**>{  **boolean** [areEqual](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, [U](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) o2)} | (**a**,**b**) -> a.contains(b); |

*Exception*

**Ohne try-with-recources:** finally { stream.close();

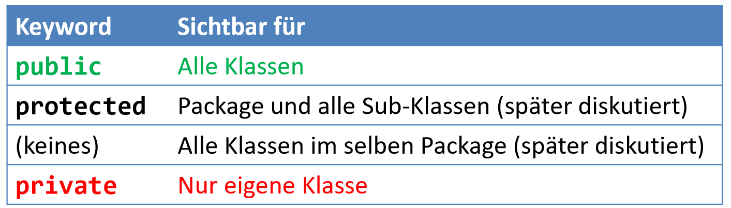
**Unchecked Exception:**

**-** Klasse, die von RuntimeException erbt, nicht behandelt werden muss

**Checked Exception:**

**-** Klasse, die von Throwable erbt, aber nicht von RuntimeException und nicht von Error

- Exception, welche in auftretenden Methoden behandelt(try-catch) oder per throws-Klausel deklariert/weitergegeben werden muss.

Zugriffskontrollen

Type-Cast

|  |  |
| --- | --- |
| **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; |

**Grundlagen**

Switch-Case

**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"); }

ForEach

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

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

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

ENUM

**private** **enum** *Level* {***LEVEL1***, ***LEVEL2***, ***LEVEL3***}

*Level* currentLevel = *Level*.***LEVEL2***;  
**if**(currentLevel == *Level*.***LEVEL1***){…}

Arrays

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

intArray.length //Ohne Klammern bei Arrays

**Arrays**.*equals*(charArray1, charArray2) //Vergleichen

**Arrays**.*sort*(personArray);  
//Klasse bedingt implements Comparable <Klasse>  
//CompareTo Methode muss implentiert sein(2. Seite)

Mehr Dimensionale Arrays

**int**[][] **intArray** = **new** **int**[2][3];

intArray.length; //Anzahl Zeilen = 2

intArray[0].length); //Ausgabe Anzahl Spalen = 3;

String

**System**.***out***.println((«Hallo».equals(«Hallo»)); //true

**String** **b** = **new** String(a); // Kopieren(oder a = b)

a += b //String oder Char anhängen

Vererbung

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

(Dynamischer) Typ prüfen - instanceof

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

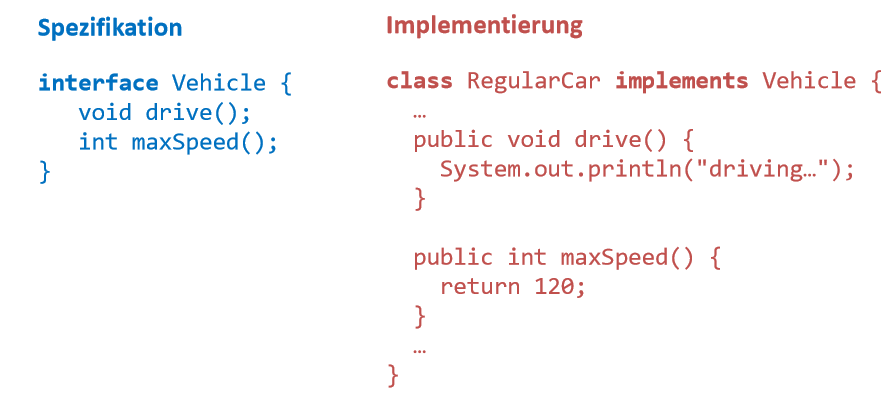
v **instanceof** **Car // True**

Abstrakte Klasse/Methode[Keine Instanzierung/nur Deklaration]

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

Final

**Eine finale Methode kann nicht überschrieben werden.  
Eine finale Klasse kann keine Subklassen haben.  
Eine finale Variabel kann nur einen Wert annehmen.**

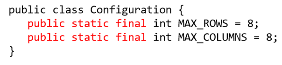
Interface

Implementierung austauschbar, Erweiterbarkeit, Unabhängige Entwicklung

Interface vs. Abstrakte Klassen

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

Konstanten

Bildschirmausschnitt**<InterfaceName || Klassenname>.KONSTANTE;** 

Exception

**try**…

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

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

**finally** {**scanner.**close();}

oder

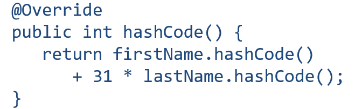
**int** **bla**(**int** **i**) throws IllegalArgumentException

**Erweiterte Funktionen**

Vergleichen von Objekten

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

**Hash Code auch überschreiben weil:**



Inkonsistenzen beim Hashing(Hashset, HashTree): Element wird allenfalls nicht gefunden. Es muss gelten: Falls x.equals(y), dann muss x.hashCode() == y.hashCode() ergeben(aber nicht umgekehrt)

Comparable Interface

Bildschirmausschnitt

Sortierung - CompareTo

***@Override*public** **int** **compareTo**(**Person** other){  
**int** **result** =lastName.compareTo(other.LastName());  
**if** (result == 0) {  
result = firstName.compareTo(other.FirstName()); }  
**return** result;}

**- <0: this ist kleiner als other**

**- >0: this ist grösser als other  
- 0: this ist gleich other**

**Collection Übersicht**

Collection Grundfunktionen

bottles.size() // Grösse der Collection

bottles.isEmpty() Prüft ob die Collection leer ist

bottles.contains(bottleGlas)Prüft ob Eleme vorhanden

ArrayList

**ArrayList**<Bottle> **bottles** = **new** ArrayList<>();   
bottles.add(bottlePlastic); Hinzufügen letzte Stelle  
bottles.add(1, bottleGlas); Alle eins nach hintenbottles.remove(0); alle eins nach vorne geschobenbottles.set(0, bottlePlastic);**// Element ersetzen**bottles.get(1) //Gibt Element der Stelle [Index]   
bottles.indexOf(bottleGlas) // Gibt Index zurück

Iterator / remove Elemente

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

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

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

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

iterator.remove();}

**else**{**System**.***out***.println(current.getMaterial());}}

LinkedList(Queue)

bottles.addLast(bottleIron);

bottles.removeFirst();  
LIFO  
bottles.addLast(bottleIron);  
bottles.removeLast();

HashSet(keine Duplikate)

**HashSet**<Bottle> **bottles** = **new** HashSet<>();

bottles.add(bottleGlas); Fügt ein Element hinzu  
bottles.remove(bottlePlastic); Entfernt Element

TreeSet(keine Duplikate, geordnet)

**TreeSet**<Bottle> **bottles** = **new** TreeSet<>();  
bottles.first() //Erstes Element(bemerke ist sortiert!)  
bottles.last() // Letztes Element

HashMap

**HashMap**<Bottle, Set<Brand>> **bottles** = **new** HashMap<>();  
bottles.put(bottlePlastic, brandsPlastic); statt add  
bottles.remove(bottleGlas); Entf. Anhand Key

bottles.get(key) Value of Key  
bottles.containsKey(bottleGlas) Key vorhanden?  
bottles.containsValue(brandsGlas) Value vorhanden?

TreeMap

**TreeMap**<Bottle, Set<Brand>> **bottles** = **new** TreeMap<>();   
bottles.firstKey();  
bottles.firstEntry();  
bottles.lastKey();  
bottles.lastEntry();

ForEach / EntrySet

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

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

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

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

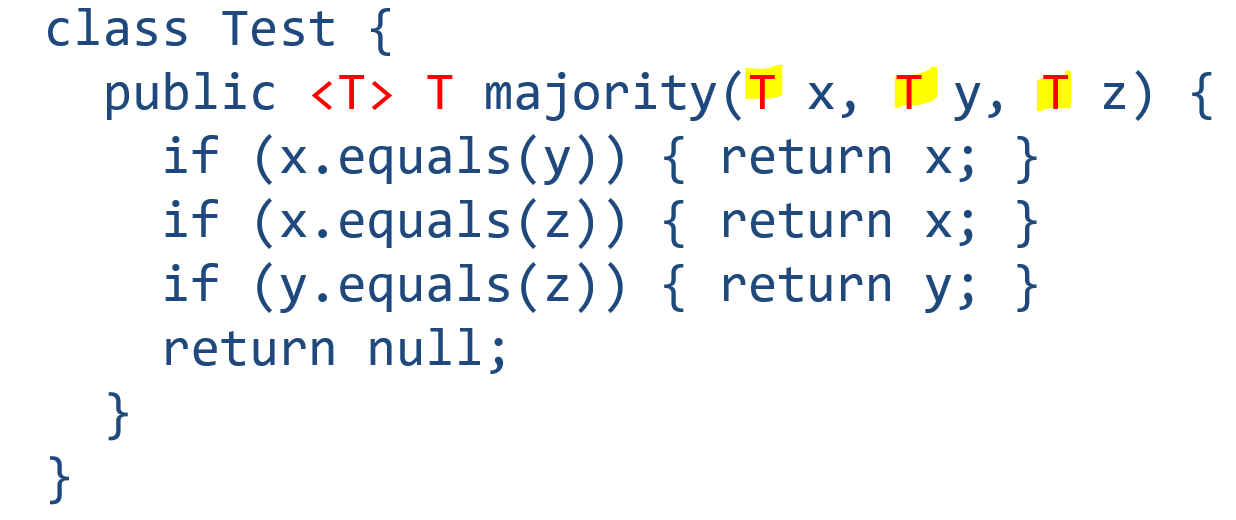
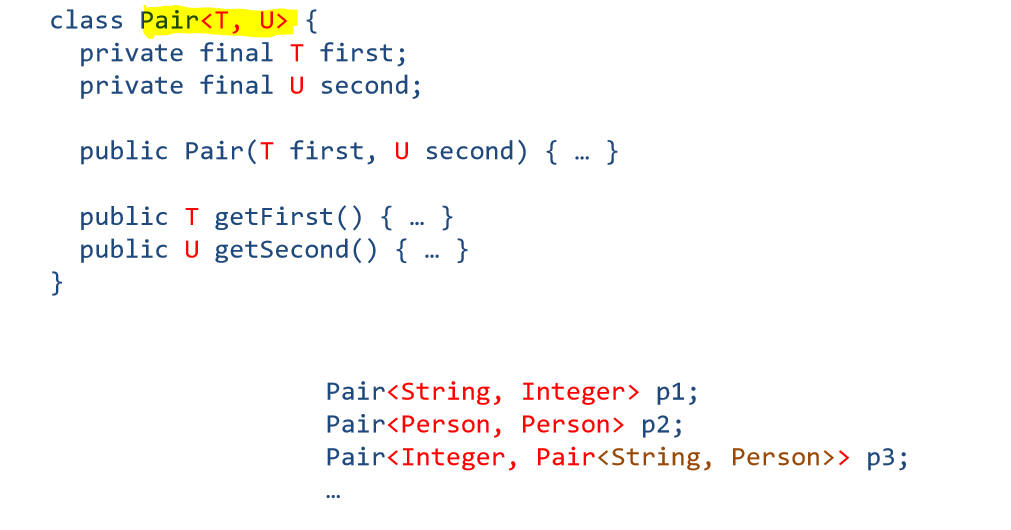
ForEach / Map.values

**for** (Set<Brand> **brands** : bottles.values()){…}

ForEach / Map.keySet()

**for** (**Bottle** **current** : bottles.keySet()){…}

**Generics , Lambdas, IO & JUnit**

Generische Klasse/Methode

Methoden Referenz

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

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

anstatt this kann auch Klassenname benutzt werden.

Lambdas

testList.sort((**t1**, **t2**) -> **Integer**.*compare*(t1.getSize(), t2.getSize()));

Comparator Bausteine

testList.sort(Comparator.*comparing*(Test::getName).thenComparing(Test::getSize).reversed());

Stream API

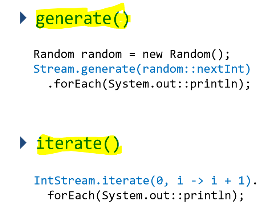
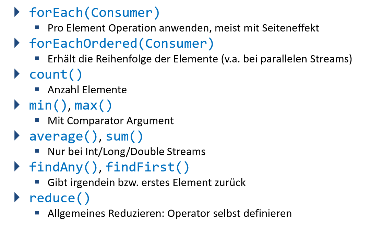
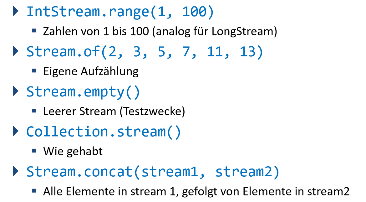
testList.stream()

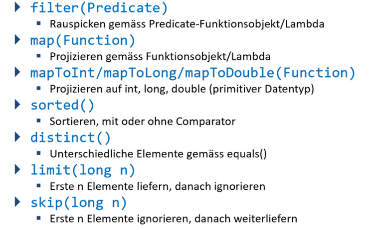
.filter(**t** -> t.getName().equals(«Susanne»))

.sorted(Comparator.*comparing*(Test::getSize).reversed())

.map(**t** -> t.getName() +" " + t.getSize())

.forEach(**System**.***out***::println);  
testList.stream()   
.mapToDouble(**t** -> t.getSize()).average()  
.ifPresent(**System**.***out***::println);

********



FileInputStream / FileOutputStream

**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;

//Implementierung mit out.write(b);

value = in.read();}

FileReader / FileWriter

**private** **static** **void** **kuerzen**(**String** inputPath, **String** outputPath) **throws** **IOException**{

**try** (**FileReader** **reader** = **new** FileReader(inputPath)){

**try** (**FileWriter** **writer** = **new** FileWriter(outputPath)){

Set<String> **kurzerText** = **new** HashSet<>();

**String** **word** = "";

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

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

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

**if** (c == '.' || c == ',' || c == ' ' || c == '\n')

{

**if** (!kurzerText.contains(word)){

kurzerText.add(word);

writer.write(word);

writer.write(c);}

**else** **if**(c != ' '){

writer.write(c);}

word = "";}

**else**{word += c;}

value = reader.read();}}}

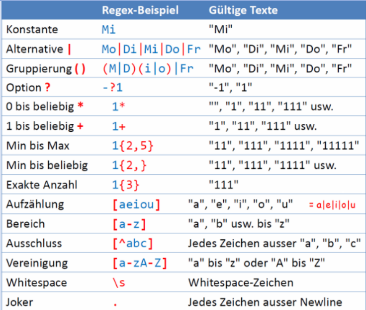
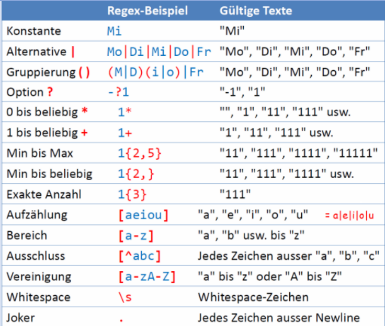
Buffered Reader

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

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

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

**while** (line != **null){…}**

Regex

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

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

**Pattern** **pattern** = **Pattern**.*compile*("[A-Za-z]\*( [A-Za-z]\*)\*( [0-9]\*)?, [0-9]{4} [A-Za-z]\*");

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

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

{

**System**.***out***.println("True: " +matcher.group(0));}

Beispiel:

"(Mo|Di|Mi|Do|Fr)\\s(0?[0-9]|1[0-9]|2[0-3]):([0-5][0-9])"

JUnit Tests

**class** **TestWallet**{

***@Test***

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

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

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

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

Gleichheitstest

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

TimeOut

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

True/False

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

wallet.deposit(100);

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

Rekursion

**List**<City> **shortestRoundtrip**(**List**<City> **visited**){

**if** (visited.size() == allCities.size()){**return** visited;}

**int** **min** = **Integer**.***MAX\_VALUE***;

**List**<City> **best** = **null**;

**for** (**City** **next** : allCities){

**if** (!visited.contains(next)){

**List**<City> **expanded** = **new** **ArrayList**<>(visited);

expanded.add(next);

**List**<City> **result** = shortestRoundtrip(expanded);

**int** **distance** = totalDistance(result);

**if** (distance < min){

min = distance;

best = result;}}}

**return** best;}

**int** **totalDistance**(**List**<City> **roundtrip**){

**int** **sum** = 0;

**for** (**int** **index** = 0; index < roundtrip.size(); index++){

**City** **current** = roundtrip.get(index);

**City** **next** = roundtrip.get((index + 1) % roundtrip.size());

sum += current.getConnections().get(next);}

**return** sum;}

Count Rekursiv

**int** **countKnowingPeople**(**Network** network, **Person** person) {

**int** **count** = 0;

**for** (**Person** **other** : network.getAllPeople()) {

**if** (other.getKnownPeople().contains(person)) {

count++;}}

**return** count;}

Rekursion mit String = + String

**Collection**<String> **generate**(**int** number) {

**Collection**<String> **result** = **new** **ArrayList**<>();

result.add(number + "");

addTerms(number, result);

addProducts(number, result);

**return** result;}

**void** **addTerms**(**int** number, **Collection**<String> result) {

**for** (**int** **term** = 1; term < number; term++) {

**for** (**String** **left** : generate(term)) {

**for** (**String** **right** : generate(number - term)) {

result.add("(" + left + " + " + right + ")");}}}}

**void** **addProducts**(**int** number, **Collection**<String> result) {

**for** (**int** **factor** = 2; factor <= number / 2; factor++) {

**if** (number % factor == 0) {

**for** (**String** **left** : generate(factor)) {

**for** (**String** **right** : generate(number / factor)) {

result.add("(" + left + " \* " + right + ")");}}}}}

}

}