# Grundlagen

Table

Description automatically generated

## Overloading vs overriding

**Overriding:** Methode mit gleicher Signatur (gleicher Name gleiche und Parameter) in Subklasse erneut implementiert

Auflösung dynamisch (dynamic Dispatch durch Laufzeitsystem)

**Overloading:** Methode mit gleichem Namen aber andere Parameter (unterschiedliche Anzahl oder Typen).

Auflösung statisch durch Compiler

## Exceptions

**Try-with-resources Beispiel:**

Scanner scanner = new Scanner(new File("hololive.txt"))

try (scanner) {  
 *// regular code*} catch (FileNotFoundException e) {  
 *// error handling*} finally {  
 *// add statements*}

Finally wird immer ausgeführt, egal ob return oder throws ausgeführt wird.

**Checked Exceptions:** Exception wird zu Laufzeit geprüft und müssen behandelt werden (catch order throws).

ClassNotFoundEx, IllegalAccessEx, IOException

**Unchecked Exceptions:** Wird vom Compiler nicht gecheckt, sie passieren wenn fehlerhafter Code ausgeführt wird. Alle unchecked Exceptions erben von RuntimeExeption.

NullPointerEx, IndexOutOfBoundsEx, ClassCastEx

**Error:** Von einem Error kann das Programm sich nicht erholen. Alle Errors sind uncheckt und können sowohl zur Laufzeit als auch zur Kompilierzeit auftreten.

OutOfMemoryError, VirtualMachineError, AssertionError

## Zugriffskontrollen

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Class | Package | Subclass (same pkg) | Subclass (same pkg) | World |
| Public | X | X | X | X | X |
| Protected | X | X | X | X |  |
| No mod. | X | X | X |  |  |
| private | x |  |  |  |  |

## Logische operationen

! 🡪 Negation ( NOT )  
&& 🡪 Logisches UND  
|| 🡪 Logisches ODER

## casting

Diagram

Description automatically generated

Logo

Description automatically generated with low confidence

Explizite Typkonversion nur zwischen numerischen Typen möglich und Informationsverlust ist möglich. Casting ist zwingend anzugeben.

int i;  
short s = (short) i;

Implizite Konversion brauchen keinen Cast und mögliche Genauigkeitsverlust.

float s = i;

## Hashing

Das Ziel von Hashing ist, dass man Elemente in einem Set / Map sehr schnell finden kann. Der Hash Algorithmus gibt einen Integer zurück, welcher auf einen Slot der Hash-Tabelle zeigt.

@Override  
public int hashCode() {  
 return id.hashCode() \* name.hashCode();  
}

## Enum

**Enum mit Konstruktor Beispiel:**

public enum Hololive {  
 *INA*(true), *KRONII*(true), *COCO*(false);  
 *//not an exhaustive list* private final boolean active;  
  
 Hololive(boolean active) {  
 this.active = active;  
 }  
  
 public boolean isActive() {  
 return active;  
 }  
}

Ein Enum ist ein eigener Datentyp mit endlichem Wertebereich. Parameter Datentyp (hier boolean) kann ersetzt weggelassen werden.

## modifiers

**Static:**   
- Verschiedene Instanzen nutzen gleiche Variable / Methoden  
- Variable ist in allen Instanzen gleich

**Final:**- Final class kann man nicht extenden  
- Final Methoden können nicht überschrieben werden  
- Final Variablen können nicht geändert werden wenn sie mal instanziert sind  
- Speziallfall: Wenn eine Final Varibale ein Objekt ist, kann sie nicht neu assigned werden, aber das Objekt auf welches gezeigt wird kann verändert werden.

## ==

**Stringvergleich mit == schlecht**

Verschiedene String-Objekte können den gleichen Text haben (z.B. bei Einlesen von I/O u.a. werden jeweils neue String Objekte angelegt). So sind die Referenzen unterschiedlich (Referenzvergleich == ergibt false), obschon die Texte der Strings dennoch gleich. Vergleich mit equals() wird daher empfohlen.

**Float und Double mit == vergleichen schlecht**

Numerische Fehler wegen der Maschinenpräzision (Repräsentation mit fixen Anzahl Stellen z.B. im IEEE 754) sind möglich und akkumulieren sich bei Operationen. So sind analytisch gleiche Formeln als numerische Floating Point Ausdrücke allenfalls ungleich (z.B. ist 0.1 + 0.1 + 0.1 != 0.3). Ein Vergleich mit einer Fehlertoleranz (Epsilon) (Math.abs(x – y) < epsilon) wird empfohlen.

# Abstraktion

## Abstrakte klassen

abstract class Pet {  
 private String name;  
 public abstract void animalSound();  
 public void sleep(){  
 wait(1000);  
 }  
}  
class Cat extends Pet {  
 @Override  
 public void animalSound() {  
 System.*out*.println("Nyaa");  
 }  
}

- Unvollständig implementierte Klasse

- Objekt von abstrakter Klasse nicht möglich

- Unvollständig implementierte Methoden müssen von Subklasse implementiert werden

**Hiding:** Subklasse definiert Instanzvariable mit gleichem Namen wie Superklasse neu. Zugriff auf Variable in Superklasse mit this.

## Interfaces

interface Existence {  
 int lifetimeLeft();  
  
 default String getMood () {  
 return "Pain Peko";  
 }  
}  
  
interface Animal extends Existence {  
 void animalSound();  
}  
  
class Cat implements Animal {  
 @Override  
 public void animalSound() {  
 System.*out*.println("Nyaa");  
 }  
  
 @Override  
 public int lifetimeLeft() {  
 return 69420;  
 }  
}

- Objekt von Interface nicht möglich

- Attribute sind standardmässig public, static und final

- Kann kein Contructor besitzen

- Klasse kann mehrere Intefaces gleichzeitig implementieren

## Interface vs abstrakte klasse

|  |  |
| --- | --- |
| **Interface** | **Abstrakte Klasse** |
| Mehrere Interfaces gleichzeitig implementiert | Maximal eine abstrakte Klasse |
| Abstrakte Methoden | Abstrakte Methoden und konkrete Methoden |
| Gute Erweiterungsmögichkeiten | Unabhängigkeit verhindern |
| Kann nur Interfaces erweitern | Kann andere Klassen erweitern und Interfaces implementieren |
| Beschreibung des Verhalten eines Klasses (Vertrag) | Basis für Subklassen mit gleichen Verhalten. Subklassen müssen Verhalten vervollständigen |

# Collections

## Lists

**LinkedList:**- Verkettete Liste der Elemente  
**-** Dynamisch hinzufügbar und entfernbar  
- LIFO und FIFO möglich  
- Intern Doppelt verkettet  
 List<String>firstList = new ArrayList<>(); **ArrayList:**

ArrayList<String> stringList = new ArrayList<>();  
stringList.add(“OO”);   
stringList.add(0, “Bsys1”);   
String x = stringList.get(1);   
stringList.set(0, “Bsys2”);   
Boolean b = stringList.contains(“CN1”)  
stringList.remove(“ICth”);  
stringList.remove(1);

## sets

**HashSet:**- In Hashtabelle gespeichert  
- Elemente geben hashCode() konsistent zu equals()  
Set<String> otherSet = new HasSet<>();  
set.add(elem);  
set.addAll(list);  
set.size();  
set.remove(elem);  
set.isEmpty();  
String[] a = (String[]) set.toArray();

**TreeSet:**- In Binärbäumen gespeichert  
- Elemente implementieren Comparable und equals()  
Set<String> firstSet = new TreeSet<>();

## Maps

**HashMap:**- Mengen von Schlüssel-Wert-Paaren  
- Braucht hashcode() und equals() Methode für sinvolle Schlüssel.

Map<Integer, String> map = new HashMap(); **TreeMap:**- Mengen von Schlüssel-Wert-Paaren  
- Nach Schlüssel sortiert  
Map<Integer, String> bachelors = new TreeMap<>();  
map.put(2000, „Hello“);  
map.containsKey(key);  
map.containsValue(value);  
String x = map.get(2000);  
for (int number : map.keySet()) {  
 sout(number);  
}

## queues / wrapper

**Queue (Warteschlange):**- Verkettete Liste der Elemente  
- Dynamisch hinzufügbar und entfernbar  
- LIFO (Stack) oder FIFO (Queue) möglich  
- Intern Doppelt verkettet

Deque<String> queue = new LinkedList<>();  
queue.addLast(elem);  
queue.addFirst(elem);  
queue.removeFirst(elem);  
queue.removeLast(elem);  
**Wrapper-Klassen:**- Wertetypen Referenztypen  
- Keine primitiven Datenwerte auf dem Stack - byte, short, int, long, float, double, char  
 - Dies sind keine Objekte (keine Referenzsemantik)  
 - Sondern direkte Werte (Kopiersemantik)

## Performance / Features

Table

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

## Iterator

Mit dem Iterator können Elementer einer Liste während dem Durchlauf gelöscht oder verändert werden.

while(stringList.iterator().hasNext()) {  
 String elem = it.next();  
 if(elem.equals("Berry")) {  
 it.remove();  
 }  
}

stringList.forEachRemaining(actualList::add);

# Lambda

people.sort((Person p1, Person p2) -> {  
 return Integer.*compare*(p1.getAge(), p2.getAge());  
});

**Syntax:** (parameter1, parameter2) -> {Code Block}

Ein Lambda ist eine Referenz auf eine anonyme Methode. Lokale Variabeln können nur zugegriffen werden wenn sie final oder effectively final sind.

# Stream API

## Operationen

• filter(Predicate) Rauspicken gem. Predicate

• map(Function) Projizieren gemäss Function

• mapToInt(Function) Proji. auf primitiver Typ

• sorted() Sortieren

• distinct() Duplikate werden gelöscht

• limit(long n) Erste n Elemente liefern

• skip(long n) Erste n Elemente ignorieren

## Terminaloperationen

• forEach(Consumer) Pro Element anwenden

• forEachOrderd(consumer) Erhält die Reihenfolge der Elemente

• count() Anzahl Elemente

• min(), max() Mit Comparator Argument

• average(), sum() Nur bei int, long, double

• findAny() Gibt irgendein Element zurück

• findFirst() gibt erstes Element zurück

# **code Beispiele**

## Comparator

class AgeComparator implements Comparator<Person>   
 @Override  
 public int compare(Person f, Person s){  
 return Integer.*compare*(f.getAge(),  
 s.getAge());

}

## comparable

@FunctionalInterface  
interface Comparable<T> {  
 int compareTo(T other); }  
  
class Person implements Comparable<Person> {  
 private int age;  
 @Override  
 public int compareTo(Person other) {  
 return Integer.*compare*(age, other.age); }  
 *//oder* @Override  
 public int compareTo(Person other) {  
 int result = lastName.compareTo(other.lastName);  
 if (result == 0) {  
 result = firstName.compareTo(other.firstName); }  
 return result;}}  
 *//oder* people.sort(this::compareByAge);*//in Methode  
 //oder* (p1, p2) -> p1.compareTo(p2)

## stream api (general)

***//get Top 10 earners***

people.stream()  
 .sorted(Comparator.comparing(  
 Person::getSalary).reversed())  
 .mapToInt(p -> p.getSalary())  
 .limit(10)  
***//get Max Age from People in Hololive***people.stream()  
 .filter(p -> p.getCity().equals("Hololive"))  
 .mapToInt(p -> p.getAge())  
 .max()  
 .getAsInt();  
***//get Average Age of Male***people.stream()  
 .filter((p) -> p.getGender().equals("Male"))  
 .mapToInt(p -> p.getAge())  
 .average();  
***//Get all female Names with 3 or less Characters***people.stream()  
 .filter((p) -> p.getGender().equals("Female")  
 && p.getFirstName().length() <= 3)  
 .map(p -> p.getFirstName())  
***//Durchschnittsalter pro Ort (mittels Collector)***people.stream()  
 .collect(Collectors.groupingBy(Person::getCity,  
 Collectors.averagingInt(Person::getAge)))  
 .forEach((city, age) -> print(city + age));  
***//All Elements in Object cotained in List***catalogue.allProperties()  
 .filter(estate -> estate instanceof House)  
 .flatMap((estate ->  
 ((House)estate).getParkings().stream()));

## Stream api (reduce)

Integer sum = integers.stream()  
 .reduce(0, (a, b) -> a + b);

*// oder*Integer sum = integers.stream()  
 .reduce(0, Integer::*sum*);

## check number for prime

private static boolean isPrime(long number) {  
 for (long factor = 2; factor \* factor <= number; factor++) {  
 if (number % factor == 0) {  
 return false;  
 }  
 }  
 return true;  
}

A picture containing clipart

Description automatically generated

## vererbung

class X {}  
class Y extends X {}  
class Z extends Y {}

public class Main {  
 public static void print(Y y) {  
 System.*out*.println("y");  
 }  
 public static void print(Z z) {  
 System.*out*.println("z");  
 }  
 public static void main(String[] args) {  
 *print*((Y) new Y());*//Ausgabe: y  
 print*((Z) new Z());*//Ausgabe: z  
 print*((Y) new Z());*//Ausgabe: y  
 print*((Z) new Y());*//Ausgabe: ClassCastEx* }  
}

class Graphic {  
 void moveTo(Graphic other) {  
 System.*out*.println("Method 1");  
 }  
}  
class Circle extends Graphic {  
 void moveTo(Graphic other) {  
 System.*out*.println("Method 2");  
 }  
 void moveTo(Circle other) {  
 System.*out*.println("Method 3");  
 }  
}  
Circle c = new Circle();  
Graphic g = new Circle();

|  |  |
| --- | --- |
| c.moveTo(g); | Method 2 |
| c.moveTo(c); | Method 3 |
| g.moveTo(c); | Method 2 |
| g.moveTo(g); | Method 2 |
| ((Graphic)c).moveTo(g); | Method 2 |
| ((Circle)g).moveTo(c); | Method 3 |

## matrix drehen

static int[][] rotate(int[][] pixels) {  
 int height = pixels.length;  
 int width = pixels[0].length;  
 int[][] result = new int[width][height];  
 for (int y = 0; y < height; y++) {  
 for (int x = 0; x < width; x++) {  
 result[x][height - 1 - y] = pixels[y][x];  
 }  
 }  
 return result;  
}

## String indents

protected String getIndent(int length) {  
 String text = "";  
 for (int i = 0; i < length; i++) {  
 text += " ";  
 }  
 return text;  
}

@Override  
public String toString(int indent) {  
 String text = getIndent(indent) + "text \n";  
 for (Food food : ingredients) {  
 text += food.toString(indent + 1);  
 }  
 return text;  
}

## Map operations

Map<Integer, String> map = new TreeMap<>();  
map.put(2000, "Hello");  
map.containsKey(2000);  
map.containsValue("Hello");  
String x = map.get(2000);  
for (int number : map.keySet()) {  
 System.*out*.println(number);  
}  
for (Map.Entry<Integer, String> number : map.entrySet()) {  
 System.*out*.println(number.getKey());  
}

## Switch-case

int a;  
switch (a) {  
 case 0:  
 break;  
 case 1:  
 break;  
}

Wenn ein Case leer ist, werden die darauffolgenden Cases ausgeführt.

## DEKLARATIONEN

int x = 3;

int y = 2;

double z = 2;

int i = 0;

int[] a = { 0, 1 };

|  |  |  |
| --- | --- | --- |
| **Ausdruck** | **Ergebniswert** | **Ergebnistyp** |
| -x / y | -1 | Int |
| -x / z | -1.5 | Double |
| 1 + x \* y | 7 | Int |
| i++ > 0 | False | Boolean |
| x > y && a[y] > 0 | ArrayIndexOutOfBoundsEx | Boolean |
| x < y && a[y] < 0 | False | Boolean |
| x > y || a[y] > 0 | True | Boolean |
| x < y || a[y] < 0 | ArrayIndexOutOfBoundsEx | Boolean |
| a[1] % a[0] | ArithmeticException | Int |
| a[0] % a[1] | 0 | int |

|  |  |
| --- | --- |
| **Ausdruck** | **Wert für Ausdruck == true** |
| a + a \* 2 == 12 | a = 4 |
| b / 2 == b | b = 0 |
| !c | c = false |
| d.length == d[0] | d = new int[] { 1 } |
| e != 0 && e / 2 == (e + 1) / 2 | e = jede ungerade negative int |
| f + "" != f | f = "abc" (jeder String) |
| g + 1 == g + 2 | g = Float/Double  .POSITIVE\_INFINITY |
| h != h | h = Float/Double.NaN |

|  |  |  |
| --- | --- | --- |
| **Ausdruck** | **Ergebniswert** | **Ergebnistyp** |
| 1 + 3 / 2 | 2 | Int |
| 1.0 + 3 / 2 | 2.0 | double |
| 1 + 3.0 / 2 | 2.5 | double |
| 1 + 3 / 0 | ArithmeticEx div by 0 | int |
| 1 + 3.0 / 0 | Double.POSITIVE\_INFINITY | double |
| 1 + 3.0 / 0.0 | Double.POSITIVE\_INFINITY | double |

## VAR konvertierung

Text, letter

Description automatically generated

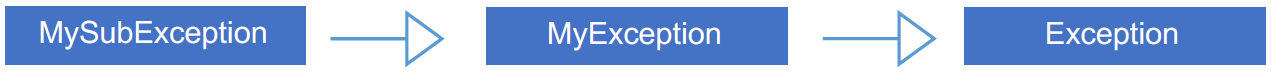
Graphical user interface, text

Description automatically generated

## To Array/collection

- Zu Array:  
Person[] p = peopleStream.toArray(Person[]:: new)   
- Zu Collection:  
List<Person> l = st.collect(Collectors.toList());

## exceptions



void throwingAndCatching(int x) throws Exception{  
 System.*out*.println("Begin");  
 try {  
 switch (x) {  
 case 0:  
 throw new MySubException();  
 case 1:  
 throw new MyException();  
 case 2:  
 *// Nothing* case 3:  
 throw new RuntimeException();  
 case 4:  
 throw new Exception();  
 }  
 } catch (MySubException e) {  
 System.*out*.println("Catch MySubExc");  
 } catch (MyException e) {  
 System.*out*.println("Catch MyExc");  
 } finally {  
 System.*out*.println("Finally");  
 }  
 System.*out*.println("End");  
}

|  |  |
| --- | --- |
| **x** | **Ausgabe auf der Konsole** |
| 0 | Begin  Catch MySubException  Finally  End |
| 1 | Begin  Catch MyException  Finally  End |
| 2 | Begin  Finally  (+Error-Ausgabe: Exception in thread "main") |
| 3 | Begin  Finally  (+Error-Ausgabe: Exception in thread "main") |
| 4 | Begin  Finally  (+Error-Ausgabe: Exception in thread "main") |
| 5 | Begin  Finally  End |

## bowling totalscore calc

Rolls is a ArrayList<Integer>

public int getTotalScore() {  
 var score = 0;  
 var i = 0;  
 for (var frame = 0; frame < 10 && i < rolls.size(); frame++) {  
 if (scoreOfRolls(rollIndex, 1) == 10) {  
 score += scoreOfRolls(i, 3);  
 rollIndex++;  
 } else if (scoreOfRolls(i, 2) == 10) {  
 score += scoreOfRolls(i, 3);  
 rollIndex += 2;  
 } else {  
 score += scoreOfRolls(i, 2);  
 rollIndex += 2;  
 }  
 }  
 return score;  
}  
  
private int scoreOfRolls(int i, int rollCount) {  
 return IntStream.*range*(i, rollIindex + rollCount)  
 .map(this::getOrZero)  
 .sum();  
}  
  
private int getOrZero(int i) {  
 if (i >= rolls.size()) {  
 return 0;  
 }  
 return rolls.get(i);  
}

## Combine lists of lists to one

List<String> collect = list.stream()  
 .flatMap(Collection::stream)  
 .collect(Collectors.*toList*());

static Set<Person> closestCommonAnchestor(Person first, Person second) {  
 Set<Person> result = anchestors(first);  
 result.retainAll(anchestors(second));  
 for (Person person : new HashSet<>(result)) {  
 result.removeAll(anchestors(person));  
 }  
 return result;  
}

## vorfahren rekursiv

Set<Person> anchestors(Person person) {  
 Set<Person> result = new HashSet<>();  
 addAnchestor(person.getFather(), result);  
 attaches(person.getMother(), result);  
 return result;  
}  
void addAnchestor(Person direct, Set<Person> result) {  
 if (direct != null) {  
 result.add(direct);  
 result.addAll(anchestors(direct));  
 }  
}

static Set<Person> closestCommonAnchestor(Person first, Person second) {  
 Set<Person> result = anchestors(first);  
 result.retainAll(anchestors(second));  
 for (Person person : new HashSet<>(result)) {  
 result.removeAll(anchestors(person));  
 }  
 return result;  
}