GRUNDLAGEN

TYP	BESCHREIBUNG	WERTE	
double	Kommaz. 64 bits	15 digits	0.1, 2e4
float	Kommaz. 32 bits	6-7 digits	0.1f, 2e4f
long	Ganzzahl 64bits	-2^{63} bis 2^{63}-1	0L
int	Ganzzahl 32bits	-2^{31} bis 2^{31}-1	1 1_1 0b1
short	Ganzzahl 16bits	-32'768 bis 32'767	
char	Textzeichen	'a','B','0','ë'	'\u0000'
byte	Ganzzahl 8bits	-128 bis 127	
boolean	Boolscher Wert	true, false	false

String mit doppelten Anführungszeichen " schreiben

Arithmetik: unär (++) vor binär +, Reihenfolge wie in der Mathematik

CASTING

Reihenfolge explizites casting (siehe Tabelle oben ohne boolean)
short, byte und char müssen untereinander explizit gecastet werden.
Explizite Typkonversion nur zwischen numerischen Typen möglich und Informationsverlust ist möglich.

```
int i;   short s = (short) i;
float s = i;
```

- Typ kann implizit auf Superklasse zugewiesen werden
- Superklasse auf Subklasse geht nicht (`ClassCastException`), ausser dynamischer Typ ist die Subklasse

```
(Superklasse) Subklasse = new SubKlasse();
```

- Hat kein Zugriff auf Subklassen Methoden die nicht überschrieben sind

OVERLOADING VS OVERRIDING

Overloading: Methode mit gleichem Namen aber andere Parameter (unterschiedliche Anzahl oder Typen).
Statische Typen der Argumente entscheiden was aufgerufen wird
Overriding: Methode mit identischer Signatur (gleicher Name gleiche und Parameter) in Subklasse erneut implementiert
Auflösung dynamisch (virtual call/ dynamic Dispatch durch Laufzeitsystem)

SICHTBARKEIT

`public`: Alle Klassen
`protected`: Packages und alle Sub-Klassen (aus anderen Packages)
`private`: Nur eigene und äussere Klasse (nicht Obj.)
`Keines`: Alle Klassen im selben Package

final Keyword
Attribute: Nicht veränderbar (`const`)

```
final int x = 10;
```

Methoden: Nicht überschreibbar (`@Override`)

```
final void test() {…}
```

Klassen: Nicht erbbar (`extends`)

```
final class Test {…}
```

CALL BY VALUE (JAVA KANN NUR CALL BY VALUE)

Call by Value: Kopie von Wert/Referenz
Call by Reference: Werte können verändert werden

EQUALS

Nicht mit == vergleichen
.equals()
equals() ist standardmässig nur Referenzvergleich, bei Strings bereits implementiert aber bei Arrays nicht.

```
class Person {
  …
  @Override
  public boolean equals(Object obj) {
    if (obj == null) {
      return false;
    }
    // prüf ob von unterschiedlichen Klassen erzeugt
    if (getClass() != obj.getClass()) {
      return false;
    }
    // nur bei Subklasse (extends)
    if (!super.equals(obj)) {
      return false;
    }
    // normal: (Instanzvariablen vergleichen)
    Person other = (Person)obj;
    return Objects.equals(firstName, other.firstName)
        && Objects.equals(lastName, other.lastName);
    // bei Subklasse:
    Student other = (Student) obj;
    return getNumber() == other.getNumber();
  }   }
```

EXCEPTIONS

```
Scanner scanner = new Scanner(new File("adr.txt"))
try (scanner) {
    // regular code
} catch (FileNotFoundException e) {
    // error handling
} finally {
    // add statements
}
```

Finally wird immer ausgeführt, egal ob `return` oder `throws` ausgeführt wird.
Exceptions erben von Object>Throwable>Exception und sind nicht Error
Checked Exceptions Exception wird zu Laufzeit/vom Compiler geprüft und müssen behandelt werden (try-catch order throws).

```
void exampleMethod() throws Exception {}
```

ClassNotFoundEx., IllegalAccessEx., IOException
Unchecked Exceptions = RuntimeException Wird vom Compiler nicht geprüft, passiert wenn fehlerhafter Code ausgeführt wird.
Keine `throws`-Deklaration / Behandlung nötig
NullPointerEx, IndexOutOfBoundsEx, ClassCastEx, IllegalArgumentEx
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

---

throws-Deklaration kann Superklasse enthalten

```
void test()throws Exception {
    throw new ClassNotFoundException("Test");
}
```

EXCEPTIONS BEHANDELN

Wenn `Exception` nicht gefangen wird (`catch`), stürzt das Programm nach `finally` direkt ab → Code nach `finally` wird nicht ausgeführt (sonst wird finally immer aufgerufen)
`Exception` in `finally` überschreibt schon generierte `Exception`
`catch` wird nur einmal aufgerufen → nur `Exceptions` im `try` Block werden gefangen.

```
try { /* regular code */ }
catch (IllegalAccessException e) { /* specific error handling for IllegalAccessException */ }
catch (Exception e) { /* error handling */ }
finally { /* cleanup */ }
```

Sobald im Beispielcode eine Klasse throws … in der Signatur hat, dann muss diese Methode beim Aufruf geprüft werden, auch bei Lambda!
Auch wenn ich im stream().filter() die Exception-Fälle ausschliesse.

```
coll.stream().filter(h -> !h.isImmune(pathogen))
.forEach( h -> {
  try {     h.infect(pathogen);
  } catch (ImmuneException e) {
     //can't happen
  })...
```

RETHROW EXCEPTIONS

`Exception` mittels (`catch`) fangen und mittels `throw` weiterleiten
Nützlich um überliegende Methoden über Fehler zu informieren nach Fehlerbehandlung
Beispiel:

```
catch (Exception e) {
    throw e;
}
```

EIGENE EXCEPTIONS

```
class StackException extends Exception {
 private static final long serialVersionUID = 1L;
 public MyExceptions() {} //optional
 public MyException(String message) {
    super(message); //optionale Methode
 }   }
```

TRY-FINALLY (TRY-WITH-RESOURCES)

Häufig ohne Catch sinnvoll um verwendete `Closeable` zu schliessen (Ressource im `finally` freizugeben), auch wenn `Exception` nicht behandelt wird. z.B. Scanner s, s.close() im finally

```
try (var s = new Scanner(System.in)) { // Alternative
  // work with s, Interface AutoCloseable
}
```

VAR KEYWORD

kann verwendet werden in
Lokale Variablen Deklaration (nicht Instanzvariablen)

```
public static void main(String[] args) {
    var x = 100;
}
```

Kann nicht verwendet werden in (Beispiele)
Generic Typen Deklaration

```
List<~~var~~> al = new ArrayList<>();
```

Deklaration mit Generic Typen

```
~~var~~<Integer> al = new ArrayList<Integer>();
```

Lambdas

```
~~var~~ obj = (a, b) -> (a + b);
```

VERERBUNG

Keine Mehrfachvererbung um Diamant-Problem zu verhindern

STATISCH / DYNAMISCHER TYP

Statischer Typ: Gemäss Variablendeklaration zur Compile-Zeit
Dynamischer Typ: Effektiver Typ der Instanz zur Laufzeit

```
Vehicle v = new Car();
var v = (Vehicle) new Car();
```

NORMALE CLASS

Default **Constructor** → Leerer **Constructor** automatisch generiert, wenn kein anderer definiert wurde, setzt alle Variablen zu dessen Default Werten.
Aufrufen von `public` / `protected`, **Methoden** / **Attributen** / **Constructor**, via. `super` → zeigt auf Instanz der Superklasse

```
super(); /* Constructor */
super(10); /* Constructor mit Argumenten */
super.test; /* Variable */
super.test(); /* Methode */
```

KLASSE & SUBKLASSE ERSTELLEN

```
public class Vehicle{ // Superklasse
    private int var1;
    public Vehicle(int var1) { // Konstruktor
        this.var1 = var1;
    }

    public int getVar1(int var){ // Methode
        return var1;
    }
// Subklasse Car
public class Car extends Vehicle{
    public Car() {…} // Konstruktor
}
public class RingBuffer<T> { // Superklasse
// T… initial ist optional:
 public RingBuffer(int capacity) {
   elements = new Object[capacity];
 }
 public RingBuffer(int capacity, T... initial) {
   elements = new Object[capacity];
   Arrays.stream(initial).forEach(this::write);
 }
 public void write(T element) {
  elements[writePosition] = element;
  writePosition++;  size++;
  if (writePosition == elements.length) {
    writePosition = 0;
  }
  if (size > elements.length) {
    size = elements.length; readPosition++;
  }
}
```

ABSTRACT CLASS

Kann `abstract` und nicht `abstract` Methoden enthalten
`abstract` Methoden müssen überschrieben werden (`@Override`)
Nicht `abstract` Methoden können überschrieben werden

Beispiel:

```
public abstract class Test {
   private String name;
   public abstract void test1();
   public void test2() {}
}
public class Test2 extends Test {
    @Override
    public void test1() {} /* Required */
    @Override
    public void test2() {} /* Optional */
}
```

- 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 Methode ist standardmässig public abstract möglich sind auch static und default Methoden.
- Objekt von Interface nicht möglich
- Attribute sind standardmässig public, static und final (Konstanten)
- Kann kein Contructor besitzen
- Klasse kann mehrere Interfaces gleichzeitig implementieren

Variablen in Interfaces nur Konstanten

```
final int test = 50;
```

Keine Instanzvariablen im Interface

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

ERBEN VON INTERFACES

Ähnlich wie `abstract`, Class kann aber nur Methoden ohne Definition enthalten.

Beispiel:

```
public interface ITestable {
    public void test1();
}
public class Test implements ITestable {
    @Override
    public void test1() {} /* Required */
}
```

ARRAY

Vorgegebene Länge (capacity), mehr Werte nicht erlaubt
Kann nur Werte des gegebenen Types enthalten
Kann Basis Datentypen enthalten (`int`, `char`)

```
// Possible array initalizations
var array = new int[] {1, 2, 3};
int array[] = {1, 2, 3};
var array = new int[3]; // Empty
int length = myArray.length; // ohne Klammern
myArray[0] = 3; // kein .set bei Array
```

Vergleichen mittels `Array.equals(a, b)`
`Arrays.deepEquals(a, b)` Geschachtelte Arrays

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.

HASHCODE

Zwei "gleiche" Objekte → gleicher HashCode
Ungleiche Objekte können aber gleichen HashCode haben
Hashcode wird zusammen mit equals definiert, konsistente Werte wie bei equals.

```
@Override
public int hashCode() {
    return Objects.hash(firstName, lastName);
    return Objects.hashcode(firstName);
}
```

COLLECTIONS

List	Folge von Elementen
Set	Menge von Elementen
Queue	Warteschlange
Map	Abbildung Schlüssel → Werte

Erlaubt nur Klassen, primitive Datentypen benötigen Wrapperklassen

WRAPPERKLASSEN

PRIMITIVER TYP	WRAPPER-KLASSE
char	Character
int	Integer
long, float, double, byte, boolean, short	Long, Float, Double, Byte, Boolean, Short

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)

ITERATOR FÜR COLLECTIONS

```
Iterator<Typ> it = stringList.iterator();
while(it.hasNext()) {
    Typ elem = it.next();
    if (!elem.equals("test")) {
        it.remove(); hinzufügen/löschen möglich
    }
}
```

Keine `ConcurrentModificationException`

LISTS

`null` ist einfügbar

LinkedList
- Verkettete Liste der Elemente
- Dynamisch hinzufügbar und entfernbar
- LIFO (stack) und FIFO (queue) möglich
- Kein Umkopieren bei `add()`, `remove()`
- Intern Doppelt verkettet (vorwärts/rückwärts)
  `List<String>firstList = new ArrayList<>();`

`add()`, `remove(anfang/ende)`: Sehr schnell (O(1))
`get()`, `set()`, `contains()`, `remove()`: Langsam (O(n))

```
List<String> firstList = new LinkedList<>();
//Gleiche Funktion wie ArrayList
```

ArrayList

```
Create ArrayList:
ArrayList<String> sList = new ArrayList<>();
List.of(array); Create list out of array
sList.add("00"); Add Element
sList.add(0, "Bsys1"); Add at position 0
String x = sList.get(1); Get at position 1
sList.set(0, "Bsys2"); Replace at position 0
Check if list contains element:
boolean b = sList.contains("CN1");
sList.remove("ICTh"); Remove element
sList.remove(1); Remove at position 1
```

STACK (LIFO)

```
var stack = new Stack<E>();
Entfernt oberstes Element vom Stack:
var popEl = stack.pop();
Fügt Element auf den Stack hinzu:
var pushedItem = stack.push(el);
var isEmpty = stack.empty();
```

QUEUES / WRAPPER (FIFO)

Queue (Warteschlange) Deque : double ended queue

```
Deque<String> queue = new LinkedList<>();
queue.addLast(elem);      |  queue.addFirst(elem);
queue.removeFirst(elem);  |  queue.removeLast(elem);
```

SETS

Keine Duplikate
Normale for-loop nicht möglich (for-each, iterator)

HashSet
- In Hashtabelle gespeichert
- Elemente geben hashCode() konsistent zu equals()

```
Set<String> otherSet = new HashSet<>();
set.add(elem); Add one element
set.addAll(list);
set.size();size of all contained elements
set.remove(elem); Remove the given element
set.isEmpty(); boolean
String[] a = (String[]) set.toArray();
set.contains(elem); boolean
```

TreeSet
- Sortiert, In Binärbäumen gespeichert
- Elemente implementieren Comparable und equals()

```
Set<String> firstSet = new TreeSet<>();
// Gleiche Funktionen wie HashSet
```

MAPS

Ähnlich wie Set
Mengen von Schlüssel-Wert-Paaren
Schlüssel müssen gleiche Regeln wie Sets erfüllen (Keine Duplikate)

HashMap
- Braucht hashCode() und equals() Methode für sinvolle Schlüssel.

```
Map<Integer, String> map = new HashMap<>();
// Gleiche Funktionen wie TreeMap
```

TreeMap
- Nach Schlüssel sortiert

```
Map<Integer, String> map1 = new TreeMap<>();
map1.put(2000, "Hello"); Add one element
map1.containsKey(2000); // boolean, key in map
map1.containsValue("Hello"); // boolean, value in map
String x = map1.get(2000); // get value of key
// Iterate through all keys in the map
for (int number : map1.keySet()) {
    System.out.println(number);
}
// Iterate through all values in the map
for (String value : map1.values()) {
    System.out.println(value);
}
```

PERFORMANCE / FEATURES

	Finden	Einfügen	Löschen
ArrayList	Langsam	Sehr schnell am Ende	Langsam
LinkedList	Langsam	Sehr schnell an Enden	Sehr schnell an Enden
HashSet	Sehr schnell	Sehr schnell	Sehr schnell
HashMap	Sehr schnell	Sehr schnell	Sehr schnell
TreeSet	Schnell	Schnell	Schnell
TreeMap	Schnell	Schnell	Schnell

finden = contains()

	Indexiert	Sortiert	Duplikate	null-Elem
ArrayList	Ja	Nein	Ja	Ja
LinkedList	Ja	Nein	Ja	Ja
HashSet	Nein	Nein	Nein	Ja
HashMap	Nein	Nein	Key: Nein	Ja
TreeSet	Nein	Ja	Nein	Nein
TreeMap	Nein	Ja	Key: Nein	Key: Nein

LAMBDA ADHOC METHODEN IMPLEMENTIERUNG

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

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

Compare

```
name1.compareTo(name2) vergleicht 2 Strings lexikographisch
Integer.compare(alter1, alter2) vergleicht int

java.util.function enthält Predicate funct.Inter
```

Filter Lambda

```
static interface Predicate {
    public boolean test(Person person);
}
public static List<Person> search(List<Person>
input, Predicate criterion) {
    var result = new ArrayList<Person>();
    for (var person : input) { // oder Iterator
       if (criterion.test(person)) { it. if(!)
          result.add(person);
       }
    }
    return result;
}
people.sort(this::compareByAge);//Methodenreferenz

ClassName enthält .search() Methode, lambda:
ClassName.search(people, p -> p.getAge() > 30);
```

STREAM API IMPORT JAVA.UTIL.STREAM.*;

Zwischenoperationen (verketten möglich)

filter(Predicate) Beispiel: people.stream().filter(p -> p.getAge() >= 18)
map(Function) Projizieren gemäss Function
mapToInt(Function) Proji. auf primitiver Typ (mapToDouble, mapToLong)
sorted() Sortieren mit oder ohne Comparator (z.B. Person::getAge)
distinct() Duplikate werden gelöscht gemäss equals
limit(long n) Erste n Elemente liefern
skip(long n) Erste n Elemente ignorieren
flatMap(Function) map aber Stream von Streams werden «flach»

Terminaloperationen (beenden die Kette)

forEach(Consumer) Beispiel: .forEach(s -> System.out.println(s));
forEachOrdered(Consumer) Erhält die Reihenfolge der Elemente
count() Anzahl Elemente (**long**)
min(), max() Mit Comparator Argument , liefert Optional-Objekte
average() Nur bei int, long, double und liefert Optional-Objekte
sum() Nur bei int, long, double
findAny() Gibt irgendein Element zurück
findFirst() gibt erstes Element zurück
allMatch(Predicate), anyMatch(..), noneMatch(..) boolean
Stream.concat(stream1, stream2)

Weiteres

`isPresent()` für Optional-Obj., boolean, ob Element vorhanden
`isEmpty()`: true wenn **kein** Element vorhanden ist
`get()`: Gibt Element, `Exception` wenn nicht vorhanden

Random random = new Random();
Stream.generate(random::nextInt).StreamOperationen();
IntStream.iterate(0, i -> i + 1).StreamOperationen()
Values und keyset für hashmap bei stream api nutzen
beachten welche Variante von stream, Stream allgemein oder IntStream

```
Optional<String> result = people.stream()
  .map(p -> p.getLastName())
  .reduce((name1, name2) -> name1 + name1);
```

Rückumwandlung (Stream to Collections)

```
Person[] arr = people.toArray(Person[]::new);
List<Person> list = people.toList();
```

Zu beliebiger Collection

```
HashSet<Person> set = people.toCollection(HashSet::new)
```

OPTIONAL-WRAPPER

```
Optionalobjekte behandeln
OptionalDouble averageAge = people.stream()
.mapToInt(p -> p.getAge()).average();
entweder so ausgeben
if (averageAge.isPresent()) {
  double result = averageAge.getAsDouble();
}
oder so mit Methodenreferenz  (ist äquivalent)
averageAge.ifPresent(System.out::println)

OptionalDouble.empty() Inexistenter Wert.
OptionalDouble.of(double value) Existenter Wert.
Falls existent, Consumer aufrufen:
optionalValue.ifPresent(Consumer)
liefert Wert oder, falls inexistent, dann other:
optionalValue.orElse(double other)
Nicht nur für Double definiert
```

STREAM API (CODE)

```
//get Top 10 earners
people.stream().sorted( Comparator.comparing(
    Person::getSalary).reversed() )
    .mapToInt(p -> p.getSalary()).limit(10)
//get Max Age from People in Rapperswil
people.stream().filter(p ->
  p.getCity().equals("Rapperswil"))
  .mapToInt(p -> p.getAge()).max()
  .getAsInt();//Optional, NoSuchElementException wenn leer
//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());
// Lambda oder Methodenreferenz möglich bei boolean
people.stream().filter(p -> p.isFemale()).count();
people.stream().filter(Person::isFemale).count();
```

```
// Alle Personen nach Alter mittels Collector
Map<Integer, List<Person>> peopleByAge =
people.stream().collect(
  Collectors.groupingBy(Person::getCity
      .averagingInt(Person::getAge)));
// Alle Personen mit Stadt
people.stream().collect(
  Collectors.groupingBy(Person::getCity))
.forEach((city, age) -> print(city + age));
//Durchschnittsalter pro Ort (mittels Collector)
people.stream().collect(
  Collectors.groupingBy( Person::getCity,
  Collectors.averagingInt(Person::getAge) ) )
.forEach((city, age) -> print(city + age));
```

COMPARABLE VERGLEICH IM OBJEKT DRIN, INTERFACE

Gibt zurück, -1, 1, 0
< 0 : this kleiner als other
> 0 : this grösser als other
0 : this gleich other

```
@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);  } }
Person::compareTo <=> (p1, p2) -> p1.compareTo(p2)
```

COMPARATOR VERGLEICH ALS EIGENE KLASSE

```
Empfohlen Comparator-Bausteine mit Methodenreferenz
people.sort(Comparator.comparing(Person::getAge));
people.sort(Comparator.comparing(Person::getCity).thenC
omparing(Person::getLastName).reversed());

Weniger empfohlen, nur wenn nötig
@FunctionalInterface
interface Comparator<T> {
    int compare(T first, T second);
}
class AgeComparator implements Comparator<Person>{
    @Override
    public int compare(Person f, Person s){
        return Integer.compare(f.getAge(),
                s.getAge());
}
Collections.sort(people, new AgeComparator());
people.sort(new AgeComparator());
```

ENUMERATIONS (ENUM)

Fast gleich zu Klasse enthält aber spezifische Werte die Klasse darstellen

Enum mit Konstruktor Beispiel:

```
public enum Weekday {
    MONDAY, TUESDAY, WEDNESDAY;
    public boolean isWeekend() {
        return this == SATURDAY || this == SUNDAY;
}  }

public enum Weekday {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
    FRIDAY, SATURDAY, SUNDAY;

    public boolean isWeekend() {
        return this == SATURDAY || this == SUNDAY;
} }
public enum Weekday {
    MONDAY(true), TUESDAY(true), WEDNESDAY(true),
    THURSDAY(true), FRIDAY(true),
    SATURDAY(false), SUNDAY(false);
    private boolean workDay;

    Weekday(boolean workDay) {// nur privater Konstruktor
        this.workDay = workDay;
    }
    public boolean isWorkDay() {
        return workDay;
} }
Weekday currentDay; // Deklaration
currentDay = Weekday.WEDNESDAY;
if (currentDay == Weekday.MONDAY) { ..}
currentDay = null;
if (currentDay == null) { … }
void getActivity(Weekday day) {
    switch (day) {
        case MONDAY:
            return "consulting";
        default:
            return "weekend";
    }
}
Weekday currentDay = …
if (currentDay.isWeekend()) { … }
```

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

NÜTZLICHE CODES

```
Integer.parseInt("1") // String zu int
 auto-unboxing:     auto-boxing:
 int d = b;         Integer b = 5;
```

LISTS, ARRAYS, COLLECTIONS

Combine lists of lists to one

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

To Array/collection

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

SWITCH CASE

```
int x = 5;
switch(x) {
    case 2:
        System.out.println("2");
        break;
    case 5:
        System.out.println("5");
        break;
    default:
        System.out.println("no match!");
}       Ohne Break → Fallthrough(Nächstes Case wird ausgeführt)

String howMany = switch(x) { // seit Java 14
    case 2 -> ("2");
    case 5 -> ("5");
    default -> ("many");
};
```

FOR LOOP

```
for (int i = 0; i < list.size(); i++) {…}
for (var el : list) {…} // oder set
for (var el : map.entrySet()) {
    var k = el.getKey();
    var v = el.getValue();
}
```

CODE SCHREIBEN

- ☐ statt for() stream api genutzt wenn sinnvoll (z.B. .sum())
- ☐ Beispielcode nutzen (z.B. für Klasse die Methode() > Klasse.Methode()
- ☐ Typen korrekt angegeben? Sichtbarkeit bei Klassen? Instanzvariabeln immer «private» deklarieren!
- ☐ Stream API korrekter Rückgabetyp
- ☐ bei loop 1. und letztes Element nicht vergessen
- ☐ bei collections wrapperklassen nutzen, nicht int sondern Integer
- ☐ exceptions beachten, auch bei lambda

CODE BEISPIELE

Main Methode

```
public static void main(String[] args) {…}
x = a ? b : c // ternary operator für Einfaches
```

Offene Parameterliste, Erlaubt beliebige Anzahl Argumente

```
static int sum(int... numbers) {
      // numbers als Array benutzen .length [i]
} // varargs, s = sum(1, 2, 3);
```

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());
}
map.values().stream().mapToInt(i -> i).sum();
```

INPUT OUTPUT

Byte Streams
- 8-Bit Daten
- Genannt: InputStream, OutputStream > Closeable

Character Stream
- 16-Bit Textzeichen (UTF-16) in Java
- Genannt: Reader, Writer

byte stream

```
byte[] data = Files.readAllBytes(Path.of("in.bin"));
Files.write(Path.of("out.bin"), data);
Datei prüfen:
var sourceFile = new File(fileNameFrom);
!sourceFile.exists() !sourceFile.isFile()
throws FileNotFoundException, IOException:
private static final int BUFFER_SIZE = 4096;
try (var in = new FileInputStream(fileNameFrom);
  var out = new FileOutputStream(fileNameTo) ) {
    var buffer = new byte[BUFFER_SIZE];
    int numRead;
    while ((numRead = in.read(buffer)) != -1) {
       out.write(buffer, 0, numRead);
    }
}
try (var in = new FileInputStream("...")) {
 int value = in.read();
 while (value >= 0) {
   byte b = (byte)value;
   // work with b
   value = in.read();
 }
 in.close();
}

var out = new FileOutputStream("test.data");
while (…) {
    byte b = …;
    out.write(b);
}
out.close();
```

Character stream

```
try (var reader = new FileReader("quotes.txt",
StandardCharsets.UTF_8)) {
          int value = reader.read();
          while (value >= 0) {
             char c = (char) value; c do something
             value = reader.read();
          }
      }
Zeilenweises Lesen:
try (var reader = new BufferedReader(new
FileReader("quotes.txt"))) {
    String line;
    while ((line = reader.readLine()) != null) {
       System.out.println("test");
    }
}

try (var writer = new FileWriter("test.txt",
  StandardCharsets.UTF_8, true)) {
    writer.write("Hello!");
    writer.write('\n');  Einzelner char
}
```

Ganze Text-Datei einlesen

List<String> lines = Files.readAllLines(Path.of("in.txt"), StandardCharset.UTF_8);

Alle Zeilen als Stream API

Stream<String> lines = Files.lines(Path.of("in.txt"), StandardCharset.UTF_8);
// mit Stream ist hier nicht ein I/O Stream (Byte-Stream oder so) gemeint

Ganze Text-Datei schreiben

Files.write(Path.of("out.txt"), lines, StandardCharsets.UTF_8);

IMPLIZITER CODE = ROT

```
public class Vehicle extends Object {
    private int speed;
    public Vehicle() {
       super();
       speed = 0;
    }
}
public class Car extends Vehicle {
    private int doors;
    public Car() {
       super();
    }
}
```