

Contents

| | |
|--|-----------|
| Nested Classes | 3 |
| Statische Attributklasse (static nested class) | 3 |
| Innere Klasse (non-static nested class) | 3 |
| Lokale Klasse | 4 |
| Anonyme Klasse | 4 |
| Initialisierung | 4 |
| Static Initialisierung | 4 |
| Non-Static Initialisierung | 5 |
| Lazy Initialisierung | 5 |
| for-Schleifen | 6 |
| old school | 6 |
| forEach (neu) | 6 |
| Varargs | 6 |
| Variable Parameterliste | 6 |
| Aufzählung (enum) | 7 |
| Definition | 7 |
| Definition mit Konstruktor | 7 |
| Generics | 8 |
| Initialisierung | 8 |
| nicht-parametrisierte Verwendung (raw type) | 9 |
| parametrisierte Verwendung | 9 |
| Eigene Generic Klasse | 10 |
| Generics & Vererbung | 10 |
| Array von Generics | 11 |
| Bounds | 11 |
| Wildcards | 11 |
| Generische Methoden | 13 |
| Autoboxing | 14 |
| Collections & Map | 14 |
| Annotations | 15 |
| @Deprecated | 15 |
| @Override | 15 |
| @SuppressWarnings | 16 |
| Selbstdefinierte Annotations | 16 |
| Lambdas | 18 |
| Functional Interfaces | 18 |

| | |
|---|-----------|
| forEach | 20 |
| Streams | 20 |
| Erneuerungen in Java von 6 - 23 | 22 |
| Java 6 | 22 |
| Java 7 | 22 |
| Java 9 & 10 | 23 |
| Java 11 | 23 |
| Java 14 | 23 |
| Java 15 | 24 |
| Java 16 | 24 |
| Java 17 | 25 |
| JUnit | 25 |
| Überprüfungsmethoden | 25 |
| Fixture | 25 |
| Hierarchische Gliederung (Suites) | 26 |
| assertThat & Matchers | 26 |
| assertAll | 27 |
| assertTimeout | 27 |
| assertThrows | 28 |
| Repeated Tests | 28 |
| Parameterized Test | 28 |
| Robustheit & Performance | 30 |
| Modifier | 30 |
| Visibility | 30 |
| Konstanten | 30 |
| Immutable Klassen | 31 |
| Gleichheit vs. Identität | 32 |
| HashCode | 33 |
| equals() vs. Comparable.compareTo() | 34 |
| Comparable vs. Comparator | 35 |
| Exceptions | 35 |
| Währungen | 36 |
| Performance | 37 |
| Objekterzeugung | 37 |
| Object Caching | 37 |
| Memory Leaks | 38 |
| Zeit messen | 38 |
| Serialisierung | 38 |
| Serializable Interface | 39 |
| Default Serialisierung | 39 |
| Modifizierte Serialisierung | 39 |

Externalisierung 41

- trivia
- iterable bzw. interfaces allgemein
- comparator allgemein

Nested Classes

Statische Attributklasse (static nested class)

```
class Outer() {  
    // attr., methods ...  
    static attr; // Zugriff durch Inner möglich  
  
    static class Inner { // abstract, final / public, protected, private  
        // attr., methods ...  
    }  
}  
  
public static void main() {  
    Outer.Inner x = new Outer.Inner(); // aus Sicht einer dritten Klasse  
    Inner x = new Inner();  
}
```

- kann nur auf statische Elemente der Hüllklasse (Outer) zugreifen
- Innere Klasse kann ohne ein Obj. der Hüllklasse (Outer) instanziiert werden

Anwendung: - wenn eine Attributklasse nicht auf die Hüllklasse (non-static elements) zugreifen muss -> statisch machen - Lazy-Initialisierung

Innere Klasse (non-static nested class)

Typen: attribut, lokal, anonym

Attributklasse

```
class Innen() { // sollte private o. protected sein  
    // attr., methods ...  
}  
  
public static void main() {  
    Outer x = new Outer();  
    Outer.Inner xi = x.new Inner();  
}
```

- Für Instanziierung der Inneren Klasse ist Äußere Instanz notwendig

Lokale Klasse

- Scope einer Variable

Anonyme Klasse

- Spezialfall lokale Klasse
- muss eine Superklasse (extends) haben o. ein Interface implementieren
- Bsp: `new Type(ctor params) { {initializer} <code> } = class
Tmp extends Type {}`

```
public class Anonymus {  
    int val;  
  
    public Anonymus( int i ) {  
        val = i;  
    }  
  
    void print() {  
        System.out.println( "val = " + val );  
    }  
}  
  
new Anonymus(2).print();  
  
new Anonymus(3) { // = class Tmp extends Anonymus {}  
    final int k;  
  
    { k = 7; }  
    void print() {  
        System.out.println("Anonym: " + k);  
    }  
}.print();
```

Initialisierung

Static Initialisierung

```
public static final String NAME = "Init Demo"; // einfach  
public static final String ARCH = System.getProperty("os.arch"); // mit Funktionsaufruf  
  
// Statischer Initialisierungsblock  
public static final String USER_HOME;  
static {  
    USER_HOME = System.getProperty("user.home");  
}
```

Non-Static Initialisierung

```
public String description = "Ein initialisiertes Attribut"; // einfach
public long timestamp = System.currentTimeMillis(); // mit Funktionsaufruf

// Initialisierungsblock
private String userPaths;
{
    userPaths = System.getProperty("java.class.path");
}
```

Lazy Initialisierung

- teure Obj. sollen nicht unnötig & so spät wie möglich initialisiert werden

Variante 1

```
class LazyInit {
    private FatClass fatObject;

    if (fatObject == null) {
        fatObject = new FatClass();
    }
    fatObject.doSomething();
}
```

- Problem: Zugriff auf Object erfolgt vllt. ohne Initialisierung bei vergessener Abfrage (if-Abfrage kann vergessen werden)

Variante 2

```
class LazyInitII {
    private FatClass fatObject;

    private FatClass getFatObject() {
        if (fatObject == null) {
            fatObject = new FatClass();
        }
        return fatObject;
    }

    getFatObject().doSomething();
}
```

- Vorteile: Initialisierung zentralisiert
- Problem: `getFatObject()` kann umgangen werden, Aufruf bei jeder Verwendung nötig

Variante 3 - Holder Pattern

```
class LazyInitIII {  
    private static class Holder {  
        static final FatClass fatObject = new FatClass();  
    }  
  
    Holder.fatObject.doSomething();  
}
```

- funktioniert nur mit static Attr.

for-Schleifen

old school

```
int a[] = 1, 2, 3, 4, 5;  
int sum = 0;  
  
for(int i = 0; i < a.length; i++) {  
    sum += a[i]; // readonly  
}
```

forEach (neu)

```
int a[] = 1, 2, 3, 4, 5;  
int sum = 0;  
  
for (int val : a) {  
    sum += val; // readonly  
}
```

- muss iterable implementieren
- neue Sprachfeatures werden in alten Code für Kompatibilität durch Pre-processing umgewandelt

Varargs

Variable Parameterliste

```
public static int sum(int... v) {  
    int sum = 0;  
    for (int i : v) {  
        sum += i;  
    }  
    return sum;  
}
```

```

public static void main() {
    int s1 = sum(1, 2);
    int s2 = sum(1, 1, 2, 3, 5);
    int s3 = sum();

    int[] array = {1, 2, 3, 4};
    int s4 = sum(array);
}

```

- Nur letzter Formalparameter darf Vararg-Parameter sein

Aufzählung (enum)

Definition

```

enum Seasons {
    SPRING, SUMMER, AUTUMN, WINTER;

    @Override
    public String toString() {
        if( this == SUMMER ) {
            return "Summer";
        }
        else {
            return super.toString();
        }
    }

    // um Methoden erweiterbar
    public static void main() {}
}

```

Definition mit Konstruktor

```

public enum Months {
    // Init mit Konstruktor
    JANUARY(31), FEBRUARY(28), MARCH(31), APRIL(30), MAY(31), JUNE(30), JULY(31),
    AUGUST(31), SEPTEMBER(30), OCTOBER(31), NOVEMBER(30), DECEMBER(31);

    private final int days;

    private Months(int days) {
        this.days = days;
    }
}

```

```

    public int getDays() {
        return days;
    }
}

```

Generics

- Generics erlauben es uns, eine Klasse für verschiedene Datentypen zu verwenden

Initialisierung

- nicht erlaubt:
 - `List<String> list = new List<String>();`
 - `List list = new List();`
 - `LinkedList<String> list = new List<String>();`
 - `LinkedList list = new List();`
- funktioniert, da Interface (entweder `LinkedList` oder `ArrayList` ohne down-cast)

Imports

```

import java.util.LinkedList;
import java.util.ArrayList;
import java.util.List;

```

`ArrayList<>` (mutable)

```

List<String> s1 = new ArrayList<>(Arrays.asList("ich", "bin"));
s1.add("nicht"); // allowed
s1.set(1, "nicht"); // allowed

```

Mit anonymer Klasse

```

List<String> stringListe = new ArrayList<>() {
    add("ich"); add("bin"); add("doch");
    add("nicht"); add("bloed ;-");
};

```

`List.of()` (immutable)

```

List<String> s1 = List.of("ich", "bin", "doch", "nicht", "bloed");

```

- ab JDK 9

nicht-parametrisierte Verwendung (raw type)

```
LinkedList stringListe = new LinkedList(); // generics version
stringListe.add("Java");
stringListe.add("Programmierung");
stringListe.add(new JButton("Hi")); // fuehrt spaeter zu einer ClassCastException

for (int i=0; i<stringListe.size(); i++) {
    String s = (String) stringListe.get(i); // cast mit ClassCastException
    gesamtLaenge += s.length();
}
```

- Nachteile:
 - get() gibt Object zurück (keine Typsicherheit)
 - casten notwendig -> kann zu ClassCastException führen

parametrisierte Verwendung

```
LinkedList<String> stringListe = new LinkedList<String>();
stringListe.add("Hello world"); // OK
stringListe.add(new Integer( 42 )); // Compiler-NOK
String s = stringListe.get(0); // kein Cast noetig!
```

- Typsicherheit

Beispiel (Iterable Interface)

```
import java.util.Iterator;

class List<T> implements Iterable<T> {
    public Iterator<T> iterator() {
        return new Iterator<T>() {
            private ListElem<T> iter = header;

            public boolean hasNext() {
                return iter != null;
            }

            public T next() {
                T ret = iter.data;
                iter = iter.next;
                return ret;
            }

            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }
}
```

- Schachtelung von Typen: `List<List<String>> var = new List<List<String>>>();`
- mehrere Typ-Parameter: `public interface Map<K, V> {...}`

Eigene Generic Klasse

```
class GenKlasse<T> {
    T data;
    GenKlasse(T data) {
        this.data = data;
    }
    void set(T data) {
        this.data = data;
    }
    T get() {
        return data;
    }
    public static void main(String[] args) {
        GenKlasse<String> gs = new GenKlasse<String>("Hi");
    }
}
```

- Typsicherheit zur Compilezeit, nicht Laufzeit
- Erasure: Typinformationen wird zur Laufzeit entfernt (T wird durch eigentlichen Datentyp ersetzt im kompilierten Code)

Generics & Vererbung

- `A <- B`: `class B extends A`
 - A ist Supertyp
 - B lässt sich zu A upcasten
- `List<a> <- List?:`

```
List<B> lb = new List<B>();
List<A> la = lb; // NOK
```

- `List <-> List` (bidirektional):

```
ArrayList list = new ArrayList();
ArrayList<String> s_list = list;
s_list.add("hi");

ArrayList<Integer> i_list = list;
i_list.add(new Integer(3));

int len = 0;
for (String s : s_list) {
    len += s.length(); // Runtime: Cast Integer --> String!!!
}
```

- raw types vermeiden & nicht mischen mit generics

Array von Generics

```
List<String> listen[] = new LinkedList<String>[5]; // error
List<String> listen[] = (LinkedList<String>[]) new List[5];
```

Bounds

Beschränkung des Parametertyps

- `public class List<T extends Figur> { }`
 - Figur kann Klasse, abstrakte Klasse, Interface (trotz extends) sein
 - Erasure ersetzt T durch Figur
- `public class X<T extends Number & Comparable & Iterator> { }`
 - Mehrfachbound

Wildcards

Upper Bound

- Ziel: Liste spezifizieren, die mit Number oder einer zu Number typkompatiblen Klasse (Float, Integer, ...) parametrisiert ist (upper bound)
- Nutzung: Nur Lesezugriff auf Elemente & als Parametercheck. Kein Schreibzugriff
- Kovarianz: Kann Spezialisierung verwenden (muss nicht)
- `GenTyp<? extends Number> <- GenTyp<Integer>`
- Initialisierung:

```
List <? extends Number> dExNumber;
dExNumber = new List<Number>(); // OK
dExNumber = new List<Integer>(); // OK
dExNumber = new List<String>(); // Type Mismatch
dExNumber = new List<Object>(); // Type Mismatch
dExNumber = new List(); // Warning, because of raw type
```

- Lesezugriff:

```
dExNumber = new List<Integer>(); // OK
for( Number n : dExNumber ); // OK, da Superklasse
for( Integer i : dExNumber ); // NOK, da Typ nicht bekannt
```

- Schreibzugriff:

```
dExNumber.add( new Integer(3) ); // NOK
dExNumber.contains( new Integer(3) ); // NOK
```

```
Number n = new Integer(3);
dExNumber.add(n); // NOK
```

```
dExNumber.add(null); // OK
```

- kein Schreibzugriff, da Typ nicht bekannt (kann Integer, Float, ... sein)

```
List<? extends Integer> dExInteger = new List<Integer>();
for(Integer i : dExInteger); // OK
dExInteger.add( new Integer(3) ); // NOK, da Typ unbekannt
```

- usecase - lesende Übergabe:

```
public static double sum(List<? extends Number> numberlist) {
    double sum = 0.0;
    for (Number n : numberlist) {
        sum += n.doubleValue();
    }
    return sum;
}

public static void main(String args[]) {
    List<Integer> integerList = Arrays.asList(1, 2, 3);
    System.out.println("sum = " + sum(integerList));

    List<Double> doubleList = Arrays.asList(1.2, 2.3, 3.5);
    System.out.println("sum = " + sum(doubleList));
}
```

Lower Bound

- Ziel: Liste spezifizieren, die mit Integer oder einem Supertyp von Integer parametrisiert ist (lower bound)
- Supertyp kann auch Interface sein
- Nutzung: Parameterchecks
- Kontravarianz: Kann allgemeineren Typ verwenden
- `GenTyp<? super Integer> <- GenTyp<Number>`
- Initialisierung:

```
List<? super Integer> dSupInt;
dSupInt = new ArrayList<Number>(); // OK
dSupInt = new List<Integer>(); // OK
dSupInt = new List<String>(); // Type Mismatch
dSupInt = new List<Object>(); // OK, because Object is super type of Integer
dSupInt = new List(); // Warning, because of raw type
```

- Initialisierung Interfaces:

```
dSupInt = new List<Serializable>(); // OK, da Number das Interface implementiert
dSupInt = new List<Comparable<Number>>(); // NOK, Integer implementiert nicht Comparable
dSupInt = new List<Comparable<Integer>>(); // OK, Integer implementiert Comparable
```

- Lesezugriff:

```
for(Number n : dSupInt); // NOK
for(Object o : dSupInt); // OK
```

- kein Lesezugriff, da Typ unbekannt & Liste könnte 'Object' enthalten

- Schreibzugriff:

```
dSupInt.add( new Integer(3) ); // OK, da mindestens Integer
```

```
Number ni = new Integer(3); // upcast
dSupInt.add(ni); // NOK
```

```
dSupInt.add(null); // OK
```

- usecase - schreibende Übergabe

```
// usecase example - schreibende Übergabe
public static void addCat(List<? super Cat> catList) {
    catList.add(new RedCat());
}
```

```
List<Animal> animalList= new ArrayList<Animal>();
List<Cat> catList= new ArrayList<Cat>();
List<RedCat> redCatList= new ArrayList<RedCat>();
```

```
addCat(catList);
addCat(animalList); // animal is superclass of Cat
```

```
addCat(redCatList); // NOK, because Cat is superclass of RedCat
```

Unbound

- List<?> l
- readonly
- Typ wird nie festgelegt

Generische Methoden

```
public class GenericMax {
    public static <T extends Number & Comparable<T>> T max(T... nums) { // ... = varargs
        if (nums.length == 0)
            throw new UnsupportedOperationException(
                "Does not support empty parameter list"
            );

        T max = nums[0];
        for (T n : nums)
```

```

        if (max.compareTo(n) == -1)
            max = n;
        return max;
    }

    public static void main(String[] args) {
        Integer iArr[] = {0, 0, 1, -1, 0, -2, 3, -5, 5};
        Integer imax = max(iArr);

        Double dmax = max(-2.3, 4.555, Math.PI); // Keine casts noetig
    }
}

```

Autoboxing

```

public static void main() {
    List<int> lint = new List<int>(); // NOK, da primitiv

    Liste<Integer> lInteger = new Liste<Integer>();
    lInteger.add(2);
}

```

Wrapper-Klassen:

```

int x = new Integer(5); // Autounboxing
Integer y = 6;
int z = new Integer(3) + 2;

```

- Primitive Datentypen haben Wrapper-Klassen & sind in beide Richtungen typ-kompatibel

| PDT | Wrapperklasse |
|---------|---------------|
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| char | Character |
| boolean | Boolean |
| byte | Byte |

Collections & Map

Listen:

```

ArrayList<String> list = new ArrayList<String>(); // Seq. Array
LinkedList<String> list = new LinkedList<String>(); // doppelt verkettete Liste
Vector<String> list = new Vector<String>(); // Seq. Array

```

Maps (Paare aus Schlüssel vom Typ K und Werten von Typ V (Schlüssel eindeutig)):

```

// Hashtabelle, zufällige Reihenfolge
HashMap<String, String> map = new HashMap<String, String>();
// Hashtabelle + doppelt verkettete Liste, eingefügte Reihenfolge
LinkedHashMap<String, String> map = new LinkedHashMap<String, String>();
// Rot-Schwarz-Baum, sortierte Reihenfolge
TreeMap<String, String> map = new TreeMap<String, String>();

```

Sets (jede Referenz darf nur einmal vorkommen):

```

// Hashtabelle, keine Duplikate, zufällige Reihenfolge
HashSet<String> set = new HashSet<String>();
// Hashtabelle + doppelt verkettete Liste, eingefügte Reihenfolge
LinkedHashSet<String> set = new LinkedHashSet<String>();
// Rot-Schwarz-Baum, sortierte Reihenfolge
TreeSet<String> set = new TreeSet<String>();

```

Queues:

```

LinkedList<String> queue = new LinkedList<String>(); // doppelt verkettete Liste
PriorityQueue<String> queue = new PriorityQueue<String>(); // Heap

```

Annotations

- sind Metadaten & werden direkt vor das betreffende Element geschrieben
- Nutzen: zusätzliche Semantik, Compile-Time Checks, Code Analyse durch Tools
- Syntax Zucker - keine Funktion ohne IDE/Framework
- **Methoden** von selbstdefinierten Annotation können keine Parameter haben und keine Exceptions auslösen

@Deprecated

- markiert Methode als veraltet, nur für Kompatibilität vorhanden

@Override

- Überschreibt Elemente einer Superklasse

```

public class Person {
    @Override
    public String getName() {
        return this.name;
    }
}

```

```
    }
}
```

@SuppressWarnings

- Unterdrücken Warnungen
- @SuppressWarnings("deprecation")
- @SuppressWarnings({"unused", "unchecked"})

Selbstdefinierte Annotations

Definition:

```
public @interface Auditor {}

// mit Attribut
public @interface Copyright {
    String value();
}

// mit Default Werten
public @interface Bug { // extends Annotation
    public final static String UNASSIGNED = "[N.N.]";
    public static enum CONDITION { OPEN, CLOSED }

    // Attribute von Annotation
    int id();
    String synopsis();
    String engineer() default UNASSIGNED;
    CONDITION condition() default CONDITION.OPEN; // enum
}
```

Nutzung:

```
@Copyright("Steven Burger")
public class Test { ... }
```

Meta Annotationen in java.lang.annotation

- Annotationen für Annotationen

| Annotation | Bedeutung |
|-------------|-------------------|
| @Documented | Docs erzeugen |
| @Inherited | Annotation geerbt |

| Annotation | Bedeutung |
|------------|---|
| @Retention | Beibehaltung der Annotation SOURCE =nur Source, CLASS =Bytecode, RUNTIME =Laufzeit über Reflection import java.lang.annotation.RetentionPolicy & import java.lang.annotation.Retention |
| @Target | Elemente, die annotiert werden können: TYPE (Klassen, Interfaces, Enums), ANNOTATION_TYPE (Annotations), FIELD (Attribute), PARAMETER,LOCAL_VARIABLE, CONSTRUCTOR, METHOD, PACKAGE mit import java.lang.annotation.ElementType |

komplexes Beispiel

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
public @interface Bug {
    // Constants and enums
    public final static String UNASSIGNED = "[N.N.]";
    public final static String UNDATED = "[N.N.]";
    public static enum CONDITION {OPEN, CLOSED}
    public static enum STATE {UNAPPROVED, APPROVED, ASSIGNED,
        IMPLEMENTED, TESTED, DOCUMENTED, REOPENED, WITHDRAWN,
        DUPLICATE, WILL_NOT_BE_FIXED}

    // Attributes
    int id();
    String synopsis();

    // The following attributes have defaults
    CONDITION condition() default CONDITION.OPEN;
    STATE state() default STATE.UNAPPROVED;
    String engineer() default UNASSIGNED;
    String fixedDate() default UNDATED;
}

// Einsatz
import static annotatedBugs.Bug.CONDITION;
```

```

import static annotatedBugs.Bug.STATE;

@Bug(id=378399, synopsis="Only works under Win3.11", state=STATE.WILL_NOT_BE_FIXED)
public class BugRiddled {
    int i;

    @Bug(id=339838,
        synopsis="Constructor obviously does " + "not initialize member i", state=STATE.ASSIGNED)
    void BugRiddled(int i) {
        this.i = i;
    }
}

```

Lambdas

- namenslose anonyme Funktionen
- kürzer als Verwendung von anonymer inneren Klasse
- Parametrisierung von Verhalten
- Lambda Ausdrücke werden zu funktionalen Interfaces umgewandelt
- `<Parameterliste> -> <Ausdruck> | <Block>`
- Parameter sind optional

```

(a, b) -> a + b; // Expression Lambda
(a, b) -> {
    return a + b; // Statement Lambda
}
() -> {
    System.out.println("Hello World");
}

```

Functional Interfaces

Predicate

```

import java.util.function.Predicate;

@FunctionalInterface
interface Predicate<T> {
    boolean test(T t);
}

Predicate<Person> checkAlter = p -> p.getAlter() >= 18;

```

Beispiel:

```

void example(Predicate<Person> pred) {
    if(pred.test(pElem)) {

```

```

        ...
    }
}

// mit anonymer inneren Klasse
example(
    new Predicate<Person> () {
        public boolean test(Person p) {
            return p.getAge() >= 17;
        }
    }
);

// mit Lambda
example(p -> p.getAge() >= 17);

```

Consumer

```

import java.util.function.Consumer;

@FunctionalInterface
interface Consumer<T> {
    void accept(T t);
}

Consumer<Person> printPerson = p -> System.out.println(p);

```

Beispiel:

```

void example(Predicate<Person> pred, Consumer<PhoneNumber> con) {
    if(pred.test(pElem)) {
        con.accept(pElem.getNumber());
    }
}

example(p -> p.getAge() >= 17, num -> {doSmtWithNum(num); });

```

Function

```

@FunctionalInterface
interface Function<T, R> {
    R apply(T t);
}

```

Beispiel:

```

void example(
    Predicate<Person> pred,

```

```

Function<Person, PhoneNumber> mapper,
Consumer<PhoneNumber> con) {

    if(pred.test(pElem)) {
        PhoneNumber num = mapper.apply(p);
        con.accept(num);
    }
}

example(p -> p.getAge() >= 17, p -> p.getHomePhoneNumber(), num -> {robocall(num); });
example(p -> p.getAge() >= 16, p -> p.getMobilePhoneNumber(), num -> {txtmsg(num); });

```

Supplier

```

// liefert Objekte vom Typ T
@FunctionalInterface
interface Supplier<T> {
    T get();
}

```

BinaryOperator

```

// zwei Objekte vom Typ T -> ein Objekt vom Typ T
@FunctionalInterface
interface BinaryOperator<T> {
    T apply(T t1, T t2);
}

```

forEach

- Iteriert über alle Elemente, die Iterable implementiert z.B. Collections
- erwarteter Consumer Interface

```

void forEach(Consumer action)
list.forEach(p -> System.out.println(p));

```

```

/* Referenzen auf Funktionen ab Java 8 */
list.forEach(System.out::println);

```

Streams

- Sequenz von Elementen, die nacheinander verarbeitet werden
- entspricht einer Pipeline: Source -> Operations -> Consumer
- **Sources:** Collection, Arrays, Files, ...
- **Intermediate-Operations:** filter(Predicate p), map(Function f), sorted(Comparator c)

- **Consumer/Terminal-Operations:** `sum(T)`, `collect(Collection)`, `reduce(T)`, `forEach(Consumer)`
 - `reduce`: subtotal/akkumulator, element -> subtotal + elements

Beispiel 1:

```
Integer sum = list.stream()
    .filter(num -> (num % 2) == 0)
    .map(num -> num * num)
    .reduce(0, (subt, num) -> subt + num);
```

// Mit Funktionsreferenz

```
sum = list.stream()
    .filter(num -> (num % 2) == 0)
    .map(num -> num * num)
    .reduce(0, Integer::sum);
```

Beispiel 2:

```
var list = List.of("Hello", "Java", "World");
list.stream()
    .map(word -> word.toUpperCase())
    .sorted(Comparator.comparing(word -> word.length()))
    .forEach(word -> System.out.println(word));
```

// Mit Funktionsreferenz

```
var list = List.of("Hello", "Java", "World");
list.stream()
    .map(String::toUpperCase) // unbound method reference
    .sorted(Comparator.comparing(String::length)) // unbound method reference
    .forEach(System.out::println); // bound method reference to System.out object
```

Parallel Streams

- mehrere Cores nutzen
- Voraussetzung: unabhängig von der Reihenfolge & keine Seiteneffekte
- Rückgabe Reihenfolge der Elemente kann sich ändern

Beispiel:

```
gatherPersons().parallelStream()
    .filter(p -> p.getAge() >= 18)
    .map(p -> p.getHomePhoneNumber())
    .filter(num -> !num.isOnDoNotCallList())
    .forEach(num -> { robocall(num); });
```

Erneuerungen in Java von 6 - 23

Java 6

- Diagnose und Management der VM mittels jconsole
- Integration von Java DB (Java implementierte relationale Datenbank) auf Basis von Apache Derby

Java 7

Strings in switch-Anweisungen

```
private static final String IDLE = "idle";
final String terminal = "terminated";
switch (state) {
    case "busy": // fall through works as before
    case terminal: // local final variables are o.k.
    case IDLE: // constants are o.k.
    { break; }
    default: ...
}
```

Numerische Literale

Neu: Numerische Literale dürfen Unterstriche enthalten

```
int decimal = 42;
int hex = 0x2A;
int octal = 052;
int binary = 0b101010;

long creditCardNumber = 1234_5678_9012_3456L;
long phoneNumber = +49_789_0123_45L;
long hexWords = 0xFFEC_DE5E;
```

Exception Handling - mehrere Typen erlaubt

Neu: mehrere Exceptions gleichzeitig abfangen

```
File file;
try {
    file = new File("stest.txt");
    file.createNewFile();
}
catch (final IOException | SecurityException ex) {
    System.out.println( "multiExc: " + ex );
}
```

Automatic Resource Management

Neu: try Anweisungen erzeugte Ressourcen werden autom. geschlossen, wenn das interface AutoCloseable implementiert wurde

```
try( BufferedReader br = new BufferedReader(new FileReader("test.txt")) ) {
    br.readLine();
}
catch(final IOException ex) {
}
```

Diamond Operator

Neu: Vereinfachte Schreibweise zu Instanziierung von Generics, Typ kann auf linker Seite weggelassen werden

```
Map<String, List<Integer> > map = new HashMap<>();
LinkedList<String> lls = new LinkedList<>();
```

Java 9 & 10

- Typ Inferenz:

```
var string = "Hello, World!";
var i = 42;
for(var string : strings) {}
```

- List.of("a", "b"): immutable Liste

Java 11

Typ Interferenz für Lambda Parameter: Consumer<String> printer = (var s) -> System.out.println(s);

Java 14

Erweiterung switch Statement um Arrows Syntax:

```
public String describeInt(int i) {
    String str = "not set";
    switch (i) {
        case 1, 2 -> str = "one or two";
        case 3 -> {
            str = "three" + j;
        }
    }
    return str;
}
```

- erlaubt Rückgabewert mit switch statement

Erweiterung switch Statement um yield:

```
private static String describeInt(int i) {
    return switch (i) {
        case 1, 2: yield "one or two";
        case 3: {
            System.out.println();
            yield "three";
        } // kein ";" wegen Block
        default: yield "...";
    };

    // Arrow Schreibweise
    return switch (i) {
        case 1, 2 -> "one or two";
        case 3 -> {
            System.out.println();
            yield "three";
        } // kein ";" wegen Block
        default -> "...";
    };
}
```

- erlaubt Rückgabewert mit Seiteneffekte in einem Block

Java 15

Text Blocks oder Raw Strings:

```
String string = """
    {
        "typ": "json",
        "inhalt": "Beispieltext"
    }
    """;
```

Java 16

instanceof mit Pattern Matching:

```
// vorher
if(obj instanceof String) {
    String str = (String) obj; // cast erforderlich
    System.out.println(str.length());
}

if(obj instanceof String str) {
```



```

        System.out.println(str.length()); // auto. cast
    }

```

Java 17

- Records: simplifizierte Form von Klassen
- sind immutable
- autom. Implementierungen von Methoden zum lesenden Zugriff, Konstruktor, toString(), equals(), hashCode()

```

public record PersonRecord(String name, PersonRecord partner) {
    public PersonRecord(String name) {
        this(name, null);
    }

    public String getNameInUppercase() {
        return name.toUpperCase();
    }
}

```

JUnit

- Framework Erstellung & Ausführung von Unit-Tests
- Namenskonvention: Testklasse fängt mit "Test"X an
- Getter und Setter ohne Validierungen werden meist nicht getestet („If it's too simple to break, don't test it“)
- Test-Driven Development (TDD): Testfälle vor Implementierung schreiben & testbares Design

Überprüfungsmethoden

- assertEquals(expected, actual): prüft auf Gleichheit
 - bei Object: x.equals(y)
 - bei prim. Datentypen: x == y
 - Gleitkommazahl ist Toleranz erforderlich
 - assertEquals(expected, actual, () -> "Fehlermeldung")
- assert(Not)Null(x)
- assert(Not)Same(x,y)
 - test Objektreferenz selber
- assertTrue(x), assertFalse(x)
 - bool

Fixture

- import org.junit.*: weglassen von @ort.junit vor Fixtures
- @org.junit.Test: Testmethoden

- `@org.junit.BeforeEach`: Initialisierungscode vor jeder Testmethode
- `@org.junit.AfterEach`: Aufräumcode nach jeder Testmethode
- `@org.junit.BeforeAll`: einmaliger Initialisierungscode mit static Method

```
import org.junit.*;
```

```
@BeforeAll
```

```
public static void setUpBeforeClass() throws Exception {
    System.out.println("setUpBeforeClass");
}
```

- `@org.junit.AfterAll`: einmaligen Aufräumcode

```
import org.junit.*;
```

```
@AfterAll
```

```
public static void tearDownAfterClass() throws Exception {
    System.out.println("tearDownAfterClass");
}
```

- `@Disabled`: Test wird ignoriert
 - `@Disabled("message")`
- `@DisplayName("name")`: für Console & Mouse Hover

Hierarchische Gliederung (Suites)

Fügt alle Testmethoden mehrerer Testklassen zusammen unter einer “Suite”

```
import org.junit.*;
```

```
@RunWith(Suite.class)
```

```
@Suite.SuiteClasses({ TestAddition.class, TestSubtraction.class })
```

```
public class TestSuite {}
```

assertThat & Matchers

- mit `assertThat` können sog. Matcher verwendet werden
- Matcher: static methods in `org.hamcrest.Matcher` class like `is()`, `not()`, `hasItem()`, `containsString()`, ...

```
import static org.hamcrest.*;
```

```
import org.hamcrest.*;
```

```
import static org.junit.*;
```

```
import org.junit.*;
```

```
assertThat(al, isA(ArrayList.class)); // Objektidentität
```

```
assertThat(al, instanceof(ArrayList.class)); // Vererbungshierarchie
```

```

assertThat(x, is(3));
assertThat(x, is(not(4)));
assertThat(responseString,
    either(containsString("color")).
    or(containsString("color"))
);
assertTrue(al.contains("abc")); // ohne Matcher
assertThat(al, hasItem("abc")); // jetzt mit Matcher

```

assertAll

Gruppierung von Assertions

```

import org.junit.*;

@Test
void groupedAssertions() {
    // alle Assertions werden ausgeführt, auch wenn eine fehlschlägt
    assertAll("addition",
        () -> assertEquals(11, Addition.addiere(5, 6)),
        () -> assertEquals(12, Addition.addiere(5, 7)));

    // Innerhalb des Codeblocks wird jedoch abgebrochen
    assertAll("both",
        () -> {
            assertEquals(1, Addition.addiere(1, -1));
            // nachfolgendes wird nur ausgeführt, wenn vorherige Assertion erfolgreich
            assertAll("addition", () -> assertEquals(3, Addition.addiere(1, 2)),
                () -> assertEquals(3, Addition.addiere(1, 2)));
        },
        () -> {} // wird ausgeführt, auch wenn vorherige Assertion fehlschlägt
    );
}

```

assertTimeout

ohne Ergebnis:

```

import org.junit.*;

@Test
void timeoutNotExceeded() {
    assertTimeout(ofMinutes(2), () -> {
        // task
    }); // check if task takes less than 2 minutes
}

```

mit String Ergebnis:

```
import org.junit.*;

@Test
void timeoutNotExceededWithResult() { // succeeds
    String actualResult = assertTimeout(ofMinutes(2),
        () -> { return "a result"; });
    assertEquals("a result", actualResult);
}
```

assertThrows

zum Abfangen & Prüfen von Exceptions:

```
import org.junit.*;

@Test
void exceptionTesting() {
    Throwable exception = assertThrows(IllegalArgumentException.class,
        () -> { throw new IllegalArgumentException("a message"); });

    System.out.println("exception message: " + exception.getMessage());
    assertEquals("a message", exception.getMessage());
}
```

Repeated Tests

```
import org.junit.*;

@RepeatedTest(10)
void repeatedTest() {
    ...
}

@RepeatedTest(value = 5, name = "Wiederholung {currentRepetition} von {totalRepetitions}")
void repeatedTestInGerman() {
    ...
}
```

Parameterized Test

- wiederholtes Ausführen des Testmethode mit verschiedenen Werten

Value Source:

```
import org.junit.*;
```

```

@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })
void palindromes(String candidate) {
    assertTrue(isPalindrome(candidate));
}

```

```

@ParameterizedTest
@ValueSource(ints = { 1, 2, 3 })
void testWithValueSource(int argument) {
    assertTrue(argument > 0 && argument < 4);
}

```

Enum Source:

```

import org.junit.*;

enum TimeUnit {
    MONTHS, WEEKS, DAYS, HOURS, MINUTES, SECONDS, MILLISECONDS, NANOSECONDS
}

```

```

@ParameterizedTest
@EnumSource(TimeUnit.class)
void testWithEnumSource(TimeUnit timeUnit) {
    assertNotNull(timeUnit);
}

```

```

@ParameterizedTest
@EnumSource(value = TimeUnit.class, mode = Mode.EXCLUDE, names = { "WEEKS", "DAYS", "HOURS" })
void testWithEnumSourceExclude(TimeUnit timeUnit) {
    assertFalse(EnumSet.of(TimeUnit.DAYS, TimeUnit.HOURS).contains(timeUnit));
    assertTrue(timeUnit.name().length() > 5);
}

```

- mode = Mode.EXCLUDE: Ausschluss bestimmter Enum-Werte
- names = { "WEEKS", "DAYS", "HOURS" }: Listet die Enum-Werte, die ausgeschlossen werden sollen (in diesem Fall WEEKS, DAYS und HOURS)

Method Source:

```

@ParameterizedTest
@MethodSource("stringProvider")
void testWithSimpleMethodSource(String argument) {
    assertNotNull(argument);
}

static Stream<String> stringProvider() { return Stream.of("foo", "bar"); }

```

```

@ParameterizedTest
@MethodSource("stringIntAndListProvider")

```

```

void testWithMultiArgMethodSource(String str, int num, List<String> list) {
    assertEquals(3, str.length());
    assertTrue(num >=1 && num <=2);
    assertEquals(2, list.size());
}

static Stream<Arguments> stringIntAndListProvider() {
    return Stream.of(
        Arguments.of("foo", 1, Arrays.asList("a", "b")),
        Arguments.of("bar", 2, Arrays.asList("x", "y"))
    );
}

```

Robustheit & Performance

- Zugriffsrechte möglichst restriktiv setzen

Modifier

```

abstract class // kann nicht instanziiert werden (z.B. nur als Oberklasse)
abstract method // placeholder

final int var // immutable, Konstante ohne Klasse
var = 5; // nur eine Zuweisung erlaubt
final method() // verhindert Override
final class // erlaubt keine Ableitung
void method(final int nr) // keine Parameteränderung

static var // existiert nur einmal im Speicher (unabhängig von Instanzen)
static method() // können nur auf static zugreifen, unabhängig von Instanzen
static class // nur nested classes -> unabhängig von äußerer Instanz
static {} // Wird nur beim ersten Laden der Klasse aufgeführt

volatile var // wird von mehreren Threads geteilt (nicht atomar)

synchronized method() // nur ein Thread kann gleichzeitig zugreifen

```

Visibility

```

public class // überall
private class // innerhalb der Klasse
protected class // innerhalb des Paketes und in Unterklassen
package class // innerhalb Package

```

Konstanten

- Konstanten := Als static final deklarierte Attribute

- Konstanten primitiver Datentypen werden zur Compilezeit substituiert
- per Konvention uppercase
- alle Instanzen einer Klasse teilen sich diese Konstanten

```
class A {
    public final static int KONST = 4711;
}

public static void main() {
    final int KONST = 4711;
}

// source code:
public void print() {
    System.out.println(A.KONST);
}

// kompiliert:
public void print() {
    System.out.println(4711);
}
```

Immutable Klassen

- Objekte, die nach der Instanziierung nicht mehr verändert werden können (z.B. String, Integer (Wrapperklassen), BigDecimal, ...)
- Erstellung:
 - Klasse **final** setzen
 - Attribute **private final** setzen
 - keiner Setter
 - DeepCopy Konstruktor
 - Getter liefern DeepCopy
- Vorteile:
 - Zeitgleicher Zugriff unproblematisch (Threadsicher)
 - guter Schlüssel fpr Maps & HashSet
 - Caching über Factory Pattern
- Nachteile:
 - Ressourcenverbrauch wegen neues Objekt & DeepCopies
 - viele Objekte
 - schlecht für große (da DeepCopies) & oft sich verändernde Objekte.

Collections

final bei Collections:

```
final List<String> list = new ArrayList<>();
list.add("rot"); // Inhalt kann geändert werden
```

```
list = new LinkedList<>(); // Fehler: Referenz kann nicht geändert werden
```

Unveränderliche Collections (Inhalt unveränderlich):

```
public final static Set<String> COLORS; // einmalige Initialisierung erlaubt
static {
    Set<String> temp = new HashSet<String>();
    temp.add("rot");
    temp.add("gruen");
    temp.add("blau");

    COLORS = java.util.Collections.unmodifiableSet(temp); // Set, List, Map etc.
    COLORS.add("gelb"); // Fehler: Inhalt kann nicht geändert werden
}
```

Gleichheit vs. Identität

- `==`: Compares the references of two objects to check if they point to the same memory location
- `equals()`: Compares the contents of two objects to check if they are logically equivalent

Beispiel:

```
String s1 = "text";
String s2 = s1;
String s3 = "text";

s1 == s2; // true, same reference
s1 == s3; // false, different reference
s1.equals(s3); // true, same text

class A {
    private int a;
    public A(int a) {
        this.a = a;
    }
}

A a1 = new A(47);
A a2 = a1;
A b = new A(11);

a1 == a2; // true -> same reference
a1.equals(a2); // true -> same reference -> same data
a1 == b; // false
a1.equals(b); // false
```



```
A a3 = new A(47);
a1 == a3; // false
a1.equals(a3); // false, da Object.equals implementiert werden muss
```

equals() Implementierung

- Contract:
 - Reflexivität: `x.equals(x)` ist immer true
 - Symmetrie: `x.equals(y) == y.equals(x)`
 - Transitivität: `x.equals(y) ^ y.equals(z) => x.equals(z)`
 - Konsistenz: `x.equals(y)` liefert immer gleiches Ergebnis, solange x und y nicht verändert werden.
 - `x.equals(null)` ist immer false
- `equals()` muss in allen Subklassen, die zusätzliche Attribute hinzufügen, überschrieben werden

`equals()` override:

```
@Override
public boolean equals(Object other) { // Parameter muss vom Typ Object sein
    if (this == other)
        // (other != null) && (null != other) nicht notwendig
        return true;
    if (!(other instanceof A))
        return false;
    A a = (A) other;
    return a.a == this.a;
}
```

Tipp zu `String.equals()`:

```
public void f(String s) {
    if (s.equals("<String>")) {} // wenn s null -> NullPointerException
    if ("<String>".equals(s)) {} // robuster, da NullPointerException vermieden wird
}
```

- `equals` darf nicht auf Arrays aufgerufen werden sondern: `java.util.Arrays.equals(Object[], Object[])`

HashCode

- `hashCode()` sollte mit `equals()` auch überschrieben werden
- Definition: gleicher `HashCode` für gleiche Objekte & unterschiedl. `Hash-Codes` für unterschiedl. Objekte -> sonst Probleme mit Maps, Sets etc.

```
public int hashCode() {
    int result = 17; // Basiswert
    result = 37 * result + a; // Hashcode für das Feld `a`
}
```

```

    return result;
}

```

- hashCode() muss in allen Subklasse, die zusätzliche Attribute hinzufügen, überschrieben werden

```

public int hashCode() {
    int result = super.hashCode(); // Basis-Hashcode von `AWithEquals`
    result = 37 * result + b; // Hashcode für das neue Feld `b`
    return result;
}

```

Beispiel Nutzung:

```

Map<BWithEquals, String> map = new HashMap<>();
BWithEquals b1 = new BWithEquals(2, 5);
BWithEquals b2 = new BWithEquals(2, 5);

map.put(b1, "Wert1");
map.get(b2); // Funktioniert nur korrekt, wenn `hashCode()` und `equals()` konsistent sind
map.put(b2, "Wert2"); // will override value

```

equals() vs. Comparable.compareTo()

- Vergleichbare Objekte können Comparable Interface erfüllen
- sollte nicht für Vergleich auf Gleichheit verwendet werden, sondern nur Sortierung o. allg. Elemente kleiner/größer

```

class Test implements Comparable<Test> {
    int d;

    @Override
    public int compareTo(T other) {
        if(this.d < o.d) {
            return - 1;
        }
        else if(this.d > o.d) {
            return 1;
        }
        return 0;

        // alternative for easy cases:
        return this.d - o.d;
    }
}

```

Comparable vs. Comparator

- Comparable is used for natural ordering/comparison within the class
- Comparator defines external logic

```
import java.util.Comparator;

class TestCompare implements Comparator<Test> {
    public int compare(Test o1, Test o2) {
        if(o1.d < o2.d) {
            return - 1;
        }
        else if(o1.d > o2.d) {
            return 1;
        }
        return 0;

        // alternative for easy cases:
        return o1.d - o2.d;
    }
}
```

Exceptions

- spezifisch wie möglich
- pro Exception eigener catch-block

```
try { ... }
catch (BindException e) {...}
catch (ConnectException e) {...}
...
```

- Nicht zu viele Exception Typen definieren -> Alternativ: Error Codes
- möglichst genau Fehlerbeschreibung

Exception vs. RuntimeException

- **Exception:** Checked Exception, die zur Compilezeit bekannt sind -> sollte behandelt werden
- **RuntimeException:** Unchecked Exception, die zur Laufzeit bekannt sind -> sollte nicht behandelt werden (z.B. NullPointerException)

Custom Exception

```
public class AnwendungsException extends Exception {
    // muss einen Konstruktor besitzen
    public AnwendungsException(String msg, Throwable t) {
        super(msg, t);
    }
}
```

```
    }
}
```

Exception Chaining

```
public void f() throws AnwendungsException {
    try {
        g();
    } catch (IOException e) {
        ex.printStackTrace(); // zeigt, dass AnwendungsException von einer ArrayIndexOutOfBoundsException
        throw new AnwendungsException("Fehler in g()", e);
    }
}
```

finally

- finally-Block wird immer ausgeführt
- Aufräumarbeiten Verhalten:

```
try { return 0; }
finally { return 1; } // 1 wird zurückgegeben
```

```
int i;
try { i = 0; return i; }
finally { i = 1; } // 0 wird zurückgegeben, da Rückgabewert bereits auf Stack liegt
```

Währungen

- Gleitpunktzahlen (float & double) speichern Werte mit begrenzter Präzision
-> Rundungsfehler:

```
double x = 0.1;
double y = 0.2;
(x + y); // 0.30000000000000004
```

```
float a = 0.1f;
float b = 0.2f;
(a * b); // 0.020000001
```

- Lösung - ganze Zahlen verwenden & Komma separat speichern:
- Alternativlösung - BigDecimal:

```
BigDecimal x = new BigDecimal("0.1");
BigDecimal y = new BigDecimal("0.2");
x.add(y); // Korrekt: 0.3
```

Performance

- -> OneNote extra Notizen
- Optimierung durch Algorithmen & Datenstrukturen

Objekterzeugung

- String sind immutable, Änderungen erzeugen neue Objekte
- Konkatenation wenn möglich mit StringBuffer o. neuer Klasse StringBuilder

```
String s = new String();
for (int i = 0; i < 10000 ; i++) {
    s += i + " "; // 3 Sekunden
}
```

```
StringBuffer sb = new StringBuffer();
for (int i = 0; i < 10000; i++) {
    sb.append(i);
    sb.append(" "); // 0,06 Sekunden
}
String s = sb.toString();
```

- niemals mit new erzeugen, immer nur direkte Zuweisung -> spart temporäre Objekte

```
String a = "Hello, world";
String b = "Hello, world"; // selbes Objekt wie a
String c = new String("Hello, world"); // anderes Objekt
String s1 = "Hello, ";
String s2 = "world";
String d = s1 + s2; // auch anderes Objekt
```

Object Caching

Erzeugen der Klasse & Methodenaufruf teuer:

```
public class WithoutCaching {
    public static void main( String[] args ) {
        for(int i = 0; i < 10; i++) {
            ExpensiveClass ec = new ExpensiveClass();
            ec.doSomething();
        }
    }
}
```

besser:

```
public class WithCaching {
    private static ExpensiveClass ec;
```

```

static {
    ec = new ExpensiveClass();
}

public static void main( String[] args ) {
    for(int i = 0; i < 10; i++) {
        ec.doSomething();
    }
}
}

```

Memory Leaks

Garbage Collection verhindert klassische Speicherlecks: Nicht mehr referenzierten, belegten Speicher -> **aber nicht unmöglich!**

```

// Referenzen können bestehen bleiben
public T pop() {
    if (size == 0)
        throw new IndexOutOfBoundsException("Stack underflow");
    return elements[--size];
}

// bessere Fassung mit "Ausnullen" der Referenzen auf die Objekte
public T pop() {
    if (size == 0)
        throw new IndexOutOfBoundsException("Stack underflow");
    T returnee = elements[--size];
    elements[size] = null;
    return returnee;
}

```

Überwachung der virtuellen Maschine am besten mit JConsole, VisualVM, JDK Mission Control oder sonstigen kommerziellen Profiling Tools.

Zeit messen

```

long startTime, endTime;
startTime = System.currentTimeMillis();
endTime = System.currentTimeMillis();
endTime - startTime;

```

Serialisierung

- Serialisierung: Object -> Byte-Array
- Deserialisierung: Byte-Array -> Object

- gespeichert wird Zustand: Typ (Klassenname), Struktur, nicht statische Attribute
 - nicht statisch, weil statische Attribute nicht Teil des Zustandes sind
- Anwendung: Persistenz/Speichern, Kommunikation zw. z.B. JVMs

Serializable Interface

- Für Serialisierung, muss `Serializable` Interface implementiert werden, sonst `NotSerializableException`
- `Serializable` ist ein Marker Interface und enthält daher keine Methoden
- Serialisierung wird durch JVM durchgeführt

```
public class Foo implements Serializable {
    public transient int x; // wird nicht serialisiert durch Schlüsselwort
}
```

Default Serialisierung

- Zentrale Klassen: `ObjectOutputStream` & `ObjectInputStream` aus `java.io.*`
- beide Streams müssen mit anderen Streams initialisiert werden z.B. `FileStream` o. `ByteArrayStream`
- Serialisierungsmethoden
 - Output: `writeObject()`, für primitive Datentypen `writeInt()`, `writeDouble()`, ...
 - Input: `readObject()`, für primitive Datentypen `readInt()`, `readDouble()`, ...

Objekte schreiben:

```
import java.io.*;

Car obj = new Car("BMW", 2000);
ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("file.ser")); // "file"
out.writeObject(obj); // Serialisierungsmethode
out.close();
```

Objekte lesen:

```
import java.io.*;

ObjectInputStream in = new ObjectInputStream(new FileInputStream("file.ser")); // "file.ser"
obj = (Car) in.readObject(); // Serialisierungsmethode, cast notwendig
in.close();
```

Modifizierte Serialisierung

- Hin und wieder ist Default-Serialisierung nicht ausreichend:

- langsam, da `writeObject()` & `readObject()` wird Reflections arbeiten
- hohe Redundanz
- wenig Kontrolle
- Spezialmethoden müssen implementiert werden

```
private void writeObject(ObjectOutputStream out) throws IOException;
private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException;
```

- Methoden als `private` deklarieren, da keine Implementierung eines Interfaces o. Methodenüberladung
- Zugriff auf Default-Serialisierungsmethoden mit `defaultWriteObject()` & `defaultReadObject()`

Beispiel:

```
import java.io.*;

public class CustomSerialization implements Serializable {
    private transient int a;
    private transient boolean b;

    public CustomSerialization(int a, boolean b) {
        this.a = a;
        this.b = b;
    }

    @Serial
    private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
        // in.defaultReadObject();
        a = in.readInt(); // Lesereihenfolge muss mit Schreibreihenfolge übereinstimmen
        b = in.readBoolean(); // Lesereihenfolge muss mit Schreibreihenfolge übereinstimmen
    }

    @Serial
    private void writeObject(ObjectOutputStream out) throws IOException {
        // out.defaultWriteObject();
        out.writeInt(a);
        out.writeBoolean(b);
    }
}
```

Beispiel mit Vererbung:

```
public class SerializableSuperClass implements Serializable {
    private String s = null;
    protected String getSuperString() { return s; }
    public void setSuperString(String s) { this.s = s; }
}
```



```

// super class gets also serialized automatically
// (when interface Serializable is inherited)
public class SerializableSubClass extends SerializableSuperClass {
    private String subS = null;

    public void getSubString() { return subS; }
    public void setSubString(String s) { subS = s; }

    @Serial
    private void readObject(ObjectInputStream oin) throws IOException, ClassNotFoundException {
        setSubString((String) oin.readObject());
    }

    @Serial
    private void writeObject(ObjectOutputStream oout) throws IOException {
        oout.writeObject(getSubString());
    }
}

```

Externalisierung