Contents

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 Hüllenklasse (Outer) zugreifen
- Innere Klasse kann ohne ein Obj. der Hüllenklasse (Outer) instanziiert werden

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

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 sit 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 Preprocessing 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> sl = new ArrayList<>(Arrays.asList("ich", "bin"));
sl.add("nicht"); // allowed
sl.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> sl = 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<b>?`:

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

- `List <-> List<B>` (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>();
```

```java
// 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):

```java
// 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:

```java
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

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

@SuppressWarning

- 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
`@Retention`	Beibehaltung der Annotation `SOURCE` =nur Source, `CLASS` =Bytecode, `RUNTIME` =Laufzeit über Reflection `import java.lang.annotation.RetentionPolicy` & `import java.lang.annotation.Retention`

Annotation	Bedeutung
`@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.ASSIGNEI
    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 -> {doSmthWithNum(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 Reihefolge 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

instaceof 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));
}
```

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

Enum Source:

```java
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:

```java
@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());
}
```

```java
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

```java
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

transient var // bei autom. Serialisierung wird die Var. übersprungen

volatile var // kein Cache, immer direkt auf Speicher schreiben & lesen (Threads)

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

Visibility

```java
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):

```java
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:

```java
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
```

```java
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

```java
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:

```java
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

```java
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

```java
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 ausgelöst wurde
        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 Machine 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
- Deserealisierung: Byte-Array -> Object
- gespeichert wird Zustand: Typ (Klassenname), Struktur, nicht statische Attribute
  - statische Attribute nicht Teil des Zustandes
- 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.ser" is stream name
);
out.writeObject(obj); // Serialisierungsmethode
out.close();
```

Objekte lesen:

```
import java.io.*;

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

Modifizierte Serialisierung

- Hin und wieder ist Default-Serialisierung nicht ausreichend:
  - langsam, da `writeObject()` & `readObject()` mit 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` deklariere, da keine Implementierung eines Interfaces o. Methodenüberladung
- Zugriff auf Default-Serialisierungsmethoden mit `defaultWriteObject()` & `defaultReadObject()`
- Durch eigene Methoden Implementierung können auch **transient** Attribute serialisiert werden

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 = Schreibreihenfolge
        b = in.readBoolean(); // Lesereihenfolge = Schreibreihenfolge
    }

    @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; }
    // readObject() & writeObject() are optional
}
```

```java
// 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());
    }
}
```

- Problem: Serialisierung der Superklasse kann nicht kontrolliert werden -> **Externalisierung**

Externalisierung

- Volle Kontrolle über serialisierte Form - Serialisierung der Superklasse steuern
- Superklasse wird nicht autom. serialisiert
- kann kombiniert mit Serializable Interface verwendet werden

Methoden zu implementieren:

```java
import java.io.Externalizable;

// muss Superklasse serialisieren
public void writeExternal(ObjectOutput out) throws IOException;
// muss Superklasse initialisieren!
public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;
```

Beispiel:

```java
import java.io.Externalizable;

public class ExternalizableSubClass
        extends SerializableSuperClass
        implements Externalizable {
    private String subString = null;
```

```
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
        setSuperString((String) in.readObject());
        setSubString((String) in.readObject());
    }

    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeObject(getSuperString());
        out.writeObject(getSubString());
    }
}
```

Caveats

Bei Singletons `readResolve()` Methode implementieren:

```
public class Highlander implements Serializable {
    private static final Highlander INSTANCE = new Highlander();
    private Highlander() {}
    public static Highlander getInstance() { return INSTANCE; }

    private Object readResolve() {
        return INSTANCE;
    }
}
```

Reflections

- Laufzeitinformationen über Klassen, Methoden, Felder, Konstruktoren
- Verwendung für Lesen o. Erzeugen/Schreiben & wenn Informationen zur Klasse nicht zur Compilezeit bekannt sind

Bestandteile von Klassen

- `java.lang.Class`-Objekt repräsentiert die **Metadaten** einer Klasse und dient als Einstiegspunkt für die Reflection
- Bestandteile: Name, Superklasse, Konstruktoren, Attribute, Methoden, Interfaces

Klassen reflektieren

Verschiedene Methoden um Klasse zu reflektieren:

```
// Reflektion auf Instanz
Object o = new String("Hello");
Class<?> cls = o.getClass();
```

```java
// Reflektion auf Klasse
Class<?> cls = String.class;

// über Name der Klasse als String
String className = "java.lang.String";
Class<?> cls = Class.forName(className);

// generische Variante
String className = "java.util.ArrayList";
Class<ArrayList> cls = (Class<ArrayList>) Class.forName(className);
ArrayList list = cls.newInstance(); // Kein Cast notwendig
```

getX() vs. getDeclaredX() Prefix für Methoden

- **get**: nur **public**, beinhaltet von Superklassen geerbte Elemente
- **getDeclared**: alle (auch **private**), nur Elemente der Klasse selbst (keine Superklasse)

Reflections Methoden

Attribute

Get attribute:

```java
Field f = cls.getField("name");
Field[] fields = cls.getFields();
// alternatives: getDeclaredField(), getDeclaredFields()
```

Get attribute information:

```java
String name = f.getName(); // name of attribute
Class<?> type = f.getType(); // type of attr., primitive types
// will return int.class or Integer.TYPE ...

// Modifikatoren
int mod = f.getModifiers(); // int - Zahl repräsentiert Modifier-Kombination
Modifier.toString(mod); // gibt "public static final" zurück

boolean isPublic = Modifier.isPublic(mod);
boolean isStatic = Modifier.isStatic(mod);
// ... is{Abstract, Final, Interface, Native, Private, Protected, Public, Static,
// Strict, Synchronized, Transient, Volatile}
```

Get & set attribute value:

```java
// Objekte
Object value = f.get(o); // get value of attribute of object o
f.set(o, val); // set value of attr. of object o
```

```java
// primitive Datentypen
int value = f.getInt(x);
f.setInt(x, 2);
```

Beispiel:

```java
static void inspiziereAttribute(Object obj) throws IllegalArgumentException,
IllegalAccessException {
    Class cls = obj.getClass();
    Field fields[] = cls.getDeclaredFields();

    for (Field f : fields) {
        f.setAccessible(true); // private Attr. zugreifbar machen
        Object val = f.get(obj);

        if (Modifier.isStatic(f.getModifiers()))
            continue;
        if (f.getType().isPrimitive()
            || f.getType().getName().equals("java.lang.String")) // primitive o. String
            ...
        else
            inspiziereAttribute(val); // Rekursion für nicht primitive Typen
    }
}
```

Konstruktoren

```java
class MyClass {
    private String message;

    public MyClass(String message) {
        this.message = message;
    }
}

Class<?> cls = MyClass.class;
// Constructor getConstructor(Class... parameterTypes);
Constructor<?> constructor = cls.getConstructor(String.class);
Constructor<?>[] constructors = cls.getConstructors();

// create instance
MyClass obj = (MyClass) constructor.newInstance("Hello, World!");
Object obj = constructor.newInstance("Hello, Reflection!");

// for private constructors
Constructor<?> constructor = cls.getDeclaredConstructor(String.class);
constructor.setAccessible(true);
```

```
Object obj = constructor.newInstance("Hello, Reflection!");
```

Methoden

```
// getMethod(String name, Class... parameterTypes)
Method m = cls.getMethod("length", String.class);
Method[] methods = cls.getMethods();

String name = m.getName(); // name of method
Class<?> returnType = m.getReturnType(); // return type of method
Class<?>[] paramTypes = m.getParameterTypes(); // parameter types of method

// invoke method
// invoke(Object, Object... args)
Object result = m.invoke(obj, "Hello, Reflection!");
Object result = m.invoke(null, "Hello, Reflection!"); // static method

// invoke private method
Method m = cls.getDeclaredMethod("length", String.class);
m.setAccessible(true); // private Method
Object result = m.invoke(obj, "Hello, Reflection!");

// throws NoSuchMethodException, IllegalAccessException InvocationTargetException
```

Beispiel:

```
static void inspiziereMethoden(Object obj) {
    Class cls = obj.getClass();
    Method methods[] = cls.getDeclaredMethods();

    for (Method m : methods) {
        Class parmTypes[] = m.getParameterTypes();
        for (Class c : parmTypes)
    }
}
```

```
public static void main(String[] args) throws SecurityException, NoSuchMethodException,
IllegalArgumentException, IllegalAccessException, InvocationTargetException {
    Class cls = Complex.class;
    Method valueOf = cls.getMethod("valueOf", double.class, double.class);

    Complex c = (Complex) valueOf.invoke(null, new Double(-1.), new Double(1.));
    Complex d = (Complex) valueOf.invoke(null, new Double(1.), new Double(-1.));

    Method add = cls.getMethod("add", Complex.class, Complex.class);
    Complex sum = (Complex) add.invoke(c, d);
```

```
}
```

Annotations

- Von jedem Element können Annotations mithilfe von Reflection gelesen werden
- `getAnnotations()` - Annotations von Superklassen dabei
- `getDeclaredAnnotations()`

JDBC

- JDBC = Java Database Connectivity
- SQL-basierte API zum Datenaustausch von Java-Programmen mit relationalen Datenbanken
- Classes/Interfaces:
  - **Driver**: DB Driver
  - **DriverManger**: registriert Treiber + baut DB Verbindung auf
  - **Connection**: Datenbankverbindung
  - **Statement**: SQL Anweisungen
  - **ResultSet**: Ergebnis SQL Abfrage in Zeilen + Spalten

Registrierung Treiber

- Über die Methode `registerDriver()` von `DriverManager`
- Wird auto. im statischen Initialisierungsblock der Driver-Klasse aufgerufen
  - Wird z.B. aktiviert von `Class.forName(drivername)`: `Class.forName("org.apache.derby.jdbc.C`
- Ab Java 6 wird der Treiber auto. geladen, wenn der DB Treiber entsprechend vorbereitet ist (ohne Driver-Klasse aufrufen)

Verbindungsaufbau per DriverManager

- über DriverManager mit Methode `getConnection()`

```
static Connection getConnection(String url, String user, String password);

Connection con = DriverManager.getConnection(
    "jdbc:derby://localhost:1527/dbname; create=true"
);
```

- **url format**: `jdbc:<subprotocol>:<subname>`
- **subprotocol**: Art der DB-Verbindung
- **subname**: Identifikator der DB
- **create=true**: Datenbank neu Anlegen, wenn nicht vorhanden

Verbindungsaufbau per DataSource

```
Context ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup("jdbc/database");
Connection con = ds.getConnection();
```

Connection

`java.sql.Connection` repräsentiert Verbindung zu Datenbank

```
Connection con = DriverManager.getConnection("jdbc:derby://localhost:1527/dbname");
```

Statements

- SQL Anweisungen werden über Statement-Objekte ausgeführt:
  - **Statement**: einfache, parameterlose SQL-Anweisungen
  - **PreparedStatement**: Vorkompilierte Anweisung, gut für mehrfache Ausführung, Parametrisierung möglich
  - **CallableStatement**: Aufruf von gespeicherten Prozeduren
- Statements werde mithilfe Connection erzeugt über die Methoden `createStatement()`, `prepareStatement()`, `prepareCall()`

Ausführung SQL-Anweisungen

- Ausführung mithilfe von Statement Objekt
  - **executeUpdate**: `INSERT`, `UPDATE`, `DELETE` - Rückgabewert: Anzahl der betroffenen Zeilen
  - **executeQuery**: `SELECT` - Rückgabewert: `ResultSet`
  - **execute**: beliebige SQL-Anweisungen - Rückgabewert: `boolean`

SQL-Syntax Beispiele:

```
Statement stmt = con.createStatement();

stmt.executeUpdate("INSERT INTO FLIGHT VALUES('LH2246',1,7,2023,7)");

stmt.executeUpdate("UPDATE FLIGHT " +
    "SET seats_available = seats_available - 1 " +
    "WHERE nr='LH2246' AND day=1 AND month=7 AND years=2023");

stmt.executeUpdate("DELETE FROM FLIGHT " + "WHERE seats_available < 1");

ResultSet rs = stmt.executeQuery("SELECT nr, seats_available"
    + " FROM flight WHERE nr='LH2246' "
    + "AND day=11 AND month=6 AND year=2023");


// con.prepareStatement(int parameterNumber, ? value)
```

```
PreparedStatement pstmt = con.prepareStatement(
    "SELECT * FROM FLIGHT WHERE nr=? AND month=? AND year=?"
);
pstmt.setString(1, "LH2246");
pstmt.setInt(2, 7);
pstmt.setInt(3, 2023);

ResultSet rs = pstmt.executeQuery();
```

ResultSet

Ergebnis von `executeQuery()`:

```
while (rs.next()) { // nächste Ergebniszeile oder previous() rückwärts
    System.out.println("ID: " + rs.getInt(1) + ", Value: " + rs.getString(2));
}
```

Referenzierung über Spaltenname oder Index ohne 0 möglich:

```
rs.getInt(1);
rs.getString(2);

rs.getInt("id");
rs.getString("value");
```

Updates über ResultSet:

```
rs.updateInt(2, 0);
rs.updateRow();
```

Ressourcenfreigabe

- Ressourcen müssen explizit freigegeben werden
- `close()` auf Connection, Statement, ResultSet

```
con.close(); // gibt auch alle Statement und ResultSet frei, reicht aus zum Schluss
stm.close(); // gibt auch ResultSet frei
rs.close();
```

- **auto-closeable Statements**: wird auto. aufgerufen nach verlassen eines try/catch-Blocks

```
// Datenbankzugriff auf JavaDB mit Derby
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class UpdatableResultSet {
```

```
    public static void main(String[] args) {

        try (
            // not necessary since Java 6
            // Class.forName("org.apache.derby.jdbc.ClientDriver");
            Connection conn = DriverManager.getConnection(
                "jdbc:derby://localhost:1527/jdbcDemoDB;create=true"
            );

            Statement stmt = conn.createStatement();) {

            stmt.execute(
                "CREATE TABLE konto (nr CHAR(10) PRIMARY KEY,
                    stand INTEGER, inhaber VARCHAR(40))"
            );

            stmt.executeUpdate(
                "INSERT INTO konto VALUES('0355380381', 6752, 'Behr Greta')"
            );

            ResultSet rs = stmt.executeQuery("SELECT * FROM konto");

            while (rs.next()) {
                System.out.printf("%10s    %10.2f   %s\n", rs.getString("nr"),
                    rs.getInt("stand") / 100., rs.getString("inhaber"));

                // 2,30 Kontofuehrungsgebuehr abzocken
                rs.updateInt(2, rs.getInt(2) - 230);
                rs.updateRow();
            }

            while (rs.previous()) {
                System.out.printf("%10s    %10.2f   %s\n", rs.getString(1),
                        rs.getInt(2) / 100., rs.getString(3));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Transaktionen

- **Auto-Commit-Modus**: für jedes SQL-Anweisung ein Commit
- Transaktionen: Zusammenfassen mehrerer SQL-Anweisungen zu einem Commit

- Connection Auto-Commit-Modus ausschalten mit `setAutoCommit(false)`
- bei Erfolg auf Connection-Objekt `commit()` aufrufen, sonst `rollback()`

Beispiel:

```
void bookBothDirections() {
    boolean success = false;
    try(Connection con = DriverManager.getConnection(
        "jdbc:derby://localhost:1527/jdbcDemoDB;"
    );
        con.setAutoCommit(false);
        Statement bookFlightStmt =  con.createStatement();) {
        bookFlightStmt.executeUpdate(..);

        success = true;
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        try {
            if (success) {
                con.commit();
            } else {
                con.rollback();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Batch-Processing

```
stmt.executeUpdate(stm1);
stmt.executeUpdate(stm2);

// Batch-Processing
stmt.addBatch(stm1);
stmt.addBatch(stm2);
stmt.executeBatch();
```

- funktioniert auch mit PreparedStatements

Metainformationen

- Strukturinformationen der DB zu Laufzeit abfragen
- Infos.: Wv. Spalten, Spaltenname, Spaltentyp, NULL etc.

```
Connection con = DriverManager.getConnection(
    "jdbc:derby://localhost:1527/jdbcDemoDB;create=true"
);
DatabaseMetaData dbmd = con.getMetaData();
dbmd.getDatabaseProductName();
dbmd.getDatabaseMajorVersion();
dbmd.getDatabaseMinorVersion();
dbmd.getDriverName();
dbmd.getDriverVersion();
dbmd.getUserName();
dbmd.getURL();
dbmd.getSQLKeywords();

Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("select * from sys.systables");
ResultSetMetaData rsmd = rs.getMetaData();
for (int i = 1; i <= rsmd.getColumnCount(); i++) {
    rsmd.getColumnLabel(i);
    rsmd.getColumnTypeName(i);
    rsmd.getColumnDisplaySize(i);
    rsmd.getColumnClassName(i);
}
```

Savepoint

voller Rollback nicht notwendig:

```
stmt1.executeUpdate();
Savepoint save1 = con.setSavepoint();
stmt2.executeUpdate();
if( cond )
    con.rollback(save1);
```

Threading & Concurrency

- Was ist ein Thread?:
  - unabhängiger Ausführungsstrang innerhalb eines Prozesses
  - teilt Systemressourcen mit anderen Threads
  - eigener Befehlszähler, eigener Stack
  - kann Zustände new, runnable, blocked, waiting
- Anwendung: Nebenläufigkeit/Parallelität, mehrere CPUs nutzen, Hintergrundaufgaben, Non-blocking I/O

Threads erzeugen

Methode 1: - Thread ableiten, wenn Vererbungshierarchie nicht benötigt wird

```java
// von java.lang.Thread ableiten
public class Thread1 extends Thread {
    public void run() { // overwrite run()
    }
    public void main () {
        final Thread1 tt = new Thread1();
        tt.start();
    }
}
```

Methode 2: - Runnable, wenn von anderer Klasse abgeleitet werden muss - Thread ist einfacher, Runnable flexibler

```java
// java.lang.Runnable implementieren
public class Runnable1 implements Runnable {
    public void run() { // implement run()
    }
    public void main() {
        final Runnable1 tr = new Runnable1();
        final Thread th = new Thread(tr);
        th.start();

        // Aufruf von nicht statischen Methoden
        Thread.getCurrentThread().staticMethod();
    }
}
```

Methode 3:

```java
// anonyme Klasse
public class Unsynchronized {
    public void methodOne() {
        Worker.doSomething(); // calculation with static var
    }

    public static void main(String[] args) {
        final Unsynchronized unSync = new Unsynchronized();

        final Thread t1 = new Thread() {
            public void run() {
                unSync.methodOne();
            }
        };

        t1.start();
        // second instance would cause unsynchronized access with unpredictable
        // results because of static var.
        t2.start();
```

```
    }
}
```

join()

- `join()`: auf die Beendigung eines anderen Threads warten

```
class JoinTheThread {
    static class JoinerThread extends Thread {
        public int result;

        @Override
        public void run() {
            result = 1;
        }

        public static void main(String[] args) throws InterruptedException {
            JoinerThread t = new JoinerThread();
            t.start();
            t.join();
            System.out.println(t.result);
        }
    }
}
```

setDaemon()

- VM kann stoppen, wenn daemon Thread infinite loop hat
- Thread wird gestoppt, wenn main() zuende geht und keine anderen normalen Threads (kein daemon Thread) laufen

```
class DaemonThread extends Thread {
    DaemonThread() {
        setDaemon(true);
    }

    @Override
    public void run() {
        int i = 0;
        while (true)
            ...
    }

    public static void main(String[] args) {
        new DaemonThread().start();
    }
}
```

Synchronisation

- **Kritischer Abschnitt**: Programmteil, in dem sich nur ein Thread zu einem Zeitpunkt befinden darf
- **Sperre (Lock)**: sperren beim betreten des kritischen Abschnitts, entsperren beim verlassen
- **Reentrant**: Kann mehrfach gesperrt werden vom gleichen Thread, muss auch mehrfach entsperrt werden

synchronized

- `synchronized`: Schlüsselwort für Methoden oder Blöcke
  - bei static Methoden: sperrt statisches class Object
  - bei normalen Methoden: sperrt Objekt/Instanz bzw. this
  - bei Blocksperre:
    - nur den Block selber
    - Objekte innerhalb des Blocks werden nicht gesperrt, außer explizit übergeben

Beispiel:

```
class Foo {
    Object obj = new Object();

    // sperrt Foo.class
    static synchronized void baz() {...}

    // sperrt this
    synchronized void bar() {...}

    void bingo() {
        // sperrt this.obj
        synchronized(obj) {...}

        // sperrt this
        synchronized(this) {...}
    }
}
```

reentrant Beispiel:

```
public synchronized void methodOne() { ... }
public synchronized void methodTwo() {
    methodOne();
}

public static void main() {
    final Foo foo = new Foo();
```

```
    final Thread t1 = new Thread() {
        @Override
        public void run() {
            Foo.methodTwo();
        }
    };
    t1.start();
}
```

- Innerhalb von `methodTwo()` wird `methodOne()` aufgerufen
- Da `methodOne()` ebenfalls `synchronized` ist und dieselbe Sperre (`this`) verwendet, tritt der Thread erneut in dieselbe Sperre ein, die er bereits hält

Explizites Lock

- `lock()`: Sperre setzen
- `unlock()`: Sperre entfernen
- support for reentrant
- `finally` nicht gefordert, aber guter Stil
- Unterschied **synchronized** vs. **explicit lock**
  - **synchronized**: einfach (auto. Freigabe nach Blockende), non invasiv
  - **explicit lock**: flexibel, beliebiger Scope, statisch & nicht statisch, invasiv

Beispiel `lock()`:

```
import java.util.concurrent.Lock;

final Lock l = new Lock1();
try {
    f.lock();
}
finally {
    f.unlock();
}
```

Beispiel `tryLock()`:

```
if (f.tryLock()) { // locks
    try { ... }
    finally {
        f.unlock();
    }
}
```

Beispiel ReentrantLock:

```java
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

final ReentrantLock lock = new ReentrantLock();
private void methodOne() {
    try {
        lock.lock();
        try {
            lock.lock();
        }
        finally {
            lock.unlock();
        }
    }
    finally {
        lock.unlock();
    }
}
```

Deadlocks

```java
// thread 1
synchronized (objA) {
    synchronized (objB) {}
}

// thread 2
synchronized (objB) {
    synchronized (objA) {}
}
```

- deadlock detection schwierig
- Locks durchnummerieren, in aufsteigender Reihenfolge sperren

Threads richtig stoppen + Sync Vars

Methode 1 (`synchronized`):

```java
static boolean stopFlag = false;

public synchronized void stopThread() {
    stopFlag = true;
}

private synchronized boolean isStopped() {
    return stopFlag;
}
```

```
public void run() {
    while (!isStopped()) {}
}
```

Methode 2 (`volatile`): - `volatile`: kein Cache, immer direkt auf Speicher schreiben & lesen

```
public void stopThread() {
    stopFlag = true;
}

public void run() {
    while (!stopFlag) {}
}
```

Signalisierung

- `Object.wait()`:
  - auf Bedingung warten
  - Sperre freigegeben & Objekt in Warteschlange eingereiht
- `Object.notify()`, `Object.notifyAll()`: Signal an wartendes Objekt senden
- `wait()` & `notify()` können nur aufgerufen werden, wenn Aufrufer Sperre hält

Beispiel (`synchronized`): - Wichtig für Producer-Consumer-Problem (Synchronisationsproblem) - Race Condition: mehrere Threads greifen auf gemeinsame Ressource zu (Queue) - Deadlock: zwei Threads warten aufeinander dadurch geht nichts mehr

```
class NotifySample {
    private final List<String> queue = new LinkedList<String>();

    private void produce(int msgId) throws InterruptedException {
        synchronized (queue) {
            while (queue.size() >= MAX_QUEUE_SIZE) {
                // gibt Sperre frei, wartet auf notify, Sperre muss neu erworben werden
                queue.wait();
            }
            queue.add("");
            queue.notifyAll(); // alle wartenden Threads benachrichtigen
        }
    }

    private void consume() throws InterruptedException {
        synchronized (queue) {
            while (queue.isEmpty()) {
```

```
                // gibt Sperre frei, wartet auf notify, Sperre muss neu erworben werden
                queue.wait();
            }
            queue.remove(0);
            queue.notifyAll(); // alle wartenden Threads benachrichtigen
        }
    }

    public static void main(String[] args) {
        final NotifySample notifySample = new NotifySample();

        final Thread producer1 = new Thread("Producer-1") {
            @Override
            public void run() {
                try {
                    for (int i = 0; i < MESSAGE_COUNT; i++) {
                        notifySample.produce(i);
                    }
                }
                catch (InterruptedException x) {
                    x.printStackTrace();
                }
            }
        };

        final Thread consumer1 = new Thread("Consumer-1") {
            @Override
            public void run() {
                try {
                    while (true) {
                        notifySample.consume();
                    }
                }
                catch (InterruptedException x) {
                    x.printStackTrace();
                }
            }
        };

        producer1.start();
        consumer1.start(); // continues running!!!
        consumer2.start(); // continues running!!!
    };
}
```

Beispiel (explicit Lock):

```java
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

private final Lock lock = new ReentrantLock();
private final Condition notFull = lock.newCondition();

private void produce(int msgId) throws InterruptedException {
    lock.lock();
    try {
        while (queue.size() >= MAX_QUEUE_SIZE) {
            notFull.await();
        }
        queue.add("");
        notFull.signalAll();
    }
    finally {
        lock.unlock();
    }
}

private void consume() throws InterruptedException {
    lock.lock();
    try {
        while (queue.isEmpty()) {
            notFull.await();
        }
        queue.remove(0);
        notFull.signalAll();
    }
    finally {
        lock.unlock();
    }
}
```

Aggregation vs. Komposition vs. Vererbung

Aggregation: Schwache *hat-eine* Beziehung (Auto hat Motor, Motor existiert ohne Auto)

Komposition: Starke *hat-eine* Beziehung (Haus hat Räume, Räume existieren nicht ohne Haus)

Vererbung: *ist-eine* Beziehung (Hund ist Tier)