

Contents

Modifier	2
Visibility	2
Nested Classes	2
Statische Attributklasse (static nested class)	2
Innere Klasse (non-static nested class)	3
Lokale Klasse	3
Anonyme Klasse	3
Initialisierung	4
Static Initialisierung	4
Non-Static Initialisierung	4
Lazy Initialisierung	4
for-Schleifen	5
old school	5
forEach (neu)	5
Varargs	6
Variable Parameterliste	6
Aufzählung (enum)	6
Definition	6
Definition mit Konstruktor	7
Generics	7
Initialisierung	7
nicht-parametrisierte Verwendung (raw type)	8
parametrisierte Verwendung	8
Eigene Generic Klasse	9
Generics & Vererbung	9
Array von Generics	10
Bounds	10
Wildcards	10
Generische Methoden	13
Autoboxing	13
Collections & Map	14
• trivia	
• iterable bzw. interfaces allgemein	

Modifier

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

final var // Konstante
final method() // verhindert Override
final class // erlaubt keine Vererbung

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
```

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 Hüllenklasse (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, da Integer nicht Comparable implementiert  
dSupInt = new List<Comparable<Integer>>(); // OK, da Integer Comparable implementiert
```

- 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)):

```
HashMap<String, String> map = new HashMap<String, String>(); // Hashtabelle, zufällige Reihenfolge
LinkedHashMap<String, String> map = new LinkedHashMap<String, String>(); // Hashtabelle + doppelte Verkettung
TreeMap<String, String> map = new TreeMap<String, String>(); // Rot-Schwarz-Baum, sortierte Reihenfolge
```

- Sets (jede Referenz darf nur einmal vorkommen):

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

- Queues:

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